Information Systems Laboratories
7047 Carroll Rd
San Diego, CA. 92122
(619) 535-9680 Fax: (619) 535-9848

Efficient Correlation Matrix Estimators for FPGA Implementation

Phase I –Basic Final Report

Contract No.: DASG60-97-M-0149
Contract Amount:59,975
Sponsor: Joe W. Ward, Contracting Officer
Effective Date of contract:2May97
Expiration Date of Contract:2Feb98

(Principal Investigator)
Amir Sarajedini
Paul M. Chau
J. Doss Halsey

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE 2 FEB 98	3. REPORT TYPE AND DATES COVERED FINAL 2 AUG 97 - 2 FEB 98

4. TITLE AND SUBTITLE Efficient Correlation Matrix Estimators for FPGA Implementation	5. FUNDING NUMBERS C-DASG60-97-M-0149
6. AUTHORS Sarajedini, Amir (NMN); Chau, Paul Ming; Halsey, James Doss	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) ISL Western Operations Division 7047 Carroll Road San Diego CA 92121	8. PERFORMING ORGANIZATION REPORT NUMBER To be assigned by monitoring organization
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Space & Strategic Defense Command P.O. Box 1500 (CSSD-CM-CK) Huntsville AL 35807-3801	10. SPONSORING / MONITORING AGENCY REPORT NUMBER To be assigned by monitoring organization
11. SUPPLEMENTARY NOTES N/A	
12a. DISTRIBUTION / AVAILABILITY STATEMENT Unlimited specific distribution will be as determined by the Contracting Officer.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Effective missile defense requires computer systems with extraordinary real-time computing capacity. Parallel architectures are necessary to provide these levels of performance. Alternative architectures can be developed by integrating the design of the numerical algorithm with the computing hardware. One such emerging technology is reconfigurable computing based on Field Programmable Gate Arrays (FPGAs). ISL has developed nonlinear operators that are easily implemented on FPGAs and can be used to implement correlators, matched filters, adaptive filters, and neural networks. In Phase I of this SBIR, ISL has implemented the nonlinear correlation estimators in FPGA processors and compared their performance in a standard application: DOA estimation. We have also considered the interconnection on multiple FPGA's and potential applications of such a reconfigurable computer. Based on this investigation, we propose recommendations for future development in a Phase II continuation of the present work.

14. SUBJECT TERMS field programmable gate arrays, adaptive computing, adaptive signal processing	15. NUMBER OF PAGES 92
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Significant Opportunity for Revolutionary Reconfigurable Processors

BMDO applications often require fast real-time signal processing on a space limited platform. The major bottleneck on real time systems is throughput rate for signal processing tasks. Target detection and recognition requires processing of images at variable precision and at variable rates (depending on the proximity of the imaging device and the target) with many of the processing tasks to be done in parallel. One can exploit this trade-off of precision and processing speed to increase throughput rate. The tool for exploiting this trade-off opportunity is *configware*, i.e. the ability to reconfigure the internal processing structure of FPGAs (Field Programmable Gate Arrays) processors and FPICs (Field Programmable Interconnection Circuits) networking. Configware combines the great flexibility of software and the significant processing speed advantage of optimized hardware.

Specific Details

The development of FPGA processors has allowed variable precision signal processing with variable throughput rates to be done on a single platform. The reduced precision is manifest in the smaller circuit sizes needed to do computations. Smaller circuits means more computation modules can be fit onto a single FPGA chip, so parallelism, and therefore throughput rate, is increased. Circuit size (precision) and parallelism (throughput rate) can be traded-off against one another to achieve acceptable level of ones while maximizing the other. Another method for optimizing signal processing tasks is to trade complexity for accuracy. This can be done by approximating the inner product operation which is ubiquitous in signal processing. Inner products are used in adaptive filtering, neural networks, and correlation matrix estimation. Toward this end, nonlinear estimators have been developed that avoid the costliest computations, i.e. multiplication, in favor of less costly computations, i.e. addition and logical operations. One task in this study has been to implement the nonlinear correlation estimators on FPGA processors. The FPGA implementation affords the ability to trade-off *dynamically* precision for throughput rate (via the circuit size/ parallelism trade-off) as well as the ability to tradeoff accuracy for throughput rate (via the choice of nonlinear correlation estimator).

The reconfiguration afforded by the FPGA allows adaptation of the algorithm and precision/parallelism to increase speed and data processing bandwidth. Thus FPGAs can be hardware customized for algorithm and situation specifics. In contrast to DSPs which are reprogrammable, FPGAs are *reconfigurable hardware*, not software. Consequently, they can be significantly faster than DSP's when multiple types of tasks are involved. Also, whereas ASIC's and DSP's have fixed input/output bandwidths, FPGA I/O pins can be dynamically reassigned, so that pin functionality can be traded off between for instance, control I/O and signal I/O. Hence FPGA has the considerable advantage of flexible I/O functionality and bandwidth.

Benefits: Potentially "The Next Netscape" Innovation

For BMDO, dynamic processor reconfiguration could enable substantial advancement in the technology for adaptive target detection and recognition capability. Initial detection and long-range tracking involves operating on very large data sets of low precision numbers. Target recognition and classification requires more complex operations on a smaller amount of high precision data. A functional switch from detection searching to lock on tracking to recognition and classification can be done on an FPGA-based architecture. When the target is far away, massive parallel processing of low precision numbers (e.g. 8 bit) using low complexity reduced accuracy algorithms is carried out. Within a threat range, the processor is dynamically reconfigured to carry out complex operations on high precision (e.g. 32 bit) numbers using a reduced number of processing channels. The *"on-the-fly"* reconfigurability of new FPGA architectural styles will enable mission environmental adaptivity for the ultimate in processing performance. The ability to dynamically reconfigure the hardware architecture, will enable truly adaptive computing and not just adaptive coefficient reprogramming. This will *revolutionize adaptive computing* and usher in the next most significant development in computing since the innovation of Netscape and Internet Explorer type internet browser software.

19980217 547

Diagram 2. Target Detection, Tracking, and Recognition at varying ranges require variable precision and processing speed and therefore in-flight reconfigurability

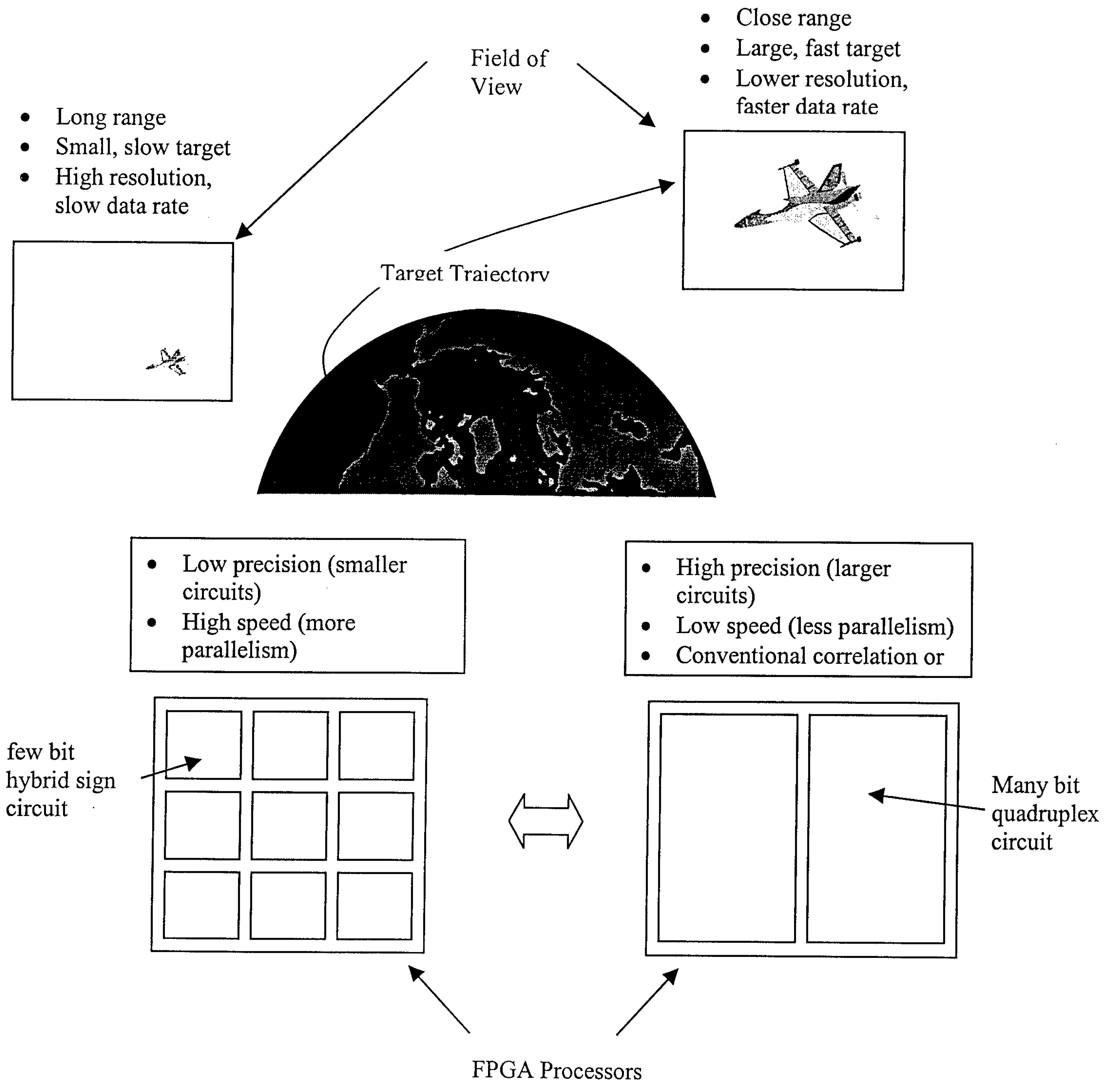
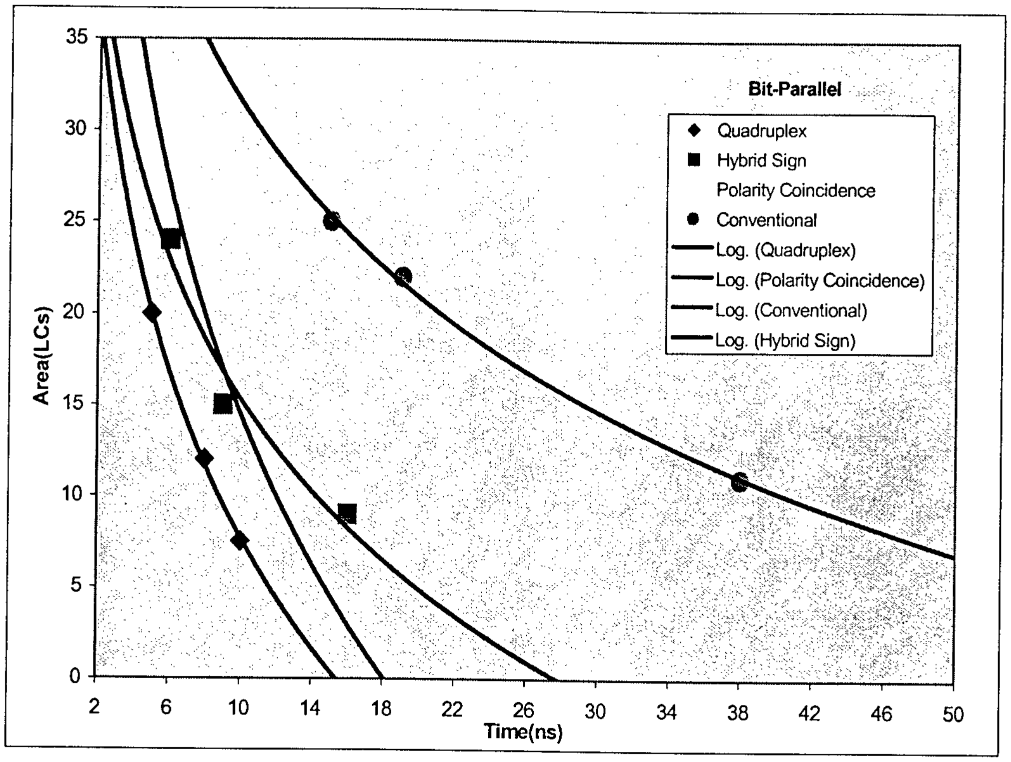

• Close range
• Large, fast target
• Lower resolution, faster data rate
Field of View
• Long range
• Small, slow target
• High resolution, slow data rate
Target Trajectory
• Low precision (smaller circuits)
• High speed (more parallelism)
• High precision (larger circuits)
• Low speed (less parallelism)
• Conventional correlation or
few bit hybrid sign circuit
Many bit quadruplex circuit
FPGA Processors

Diagram 1. Nonlinear Correlation Estimators allow speed/accuracy trade-off



- Nonlinear Correlation estimators have much higher processing speeds (3-5 times faster) and therefore higher throughput rates than the conventional correlation estimators
- FPGA reconfigurability allows area of correlation estimator to be traded-off against computation time (and therefore processing speed)

Executive Summary

Effective missile defense requires computer systems with extraordinary real-time computing capacity. Parallel architectures are necessary to provide these levels of performance. High performance computing systems can be implemented by connecting large numbers of conventional processors, but few algorithms are able to fully exploit the raw computational resources in the resulting system.

Alternative architectures can be developed by integrating the design of the numerical algorithm with the computing hardware. One such emerging technology is reconfigurable computing based on Field Programmable Gate Arrays (FPGAs). Most conventional signal processing algorithms do not map well to FPGA resources, but the same functionality can often be provided with algorithms that have been tailored to operate in a FPGA environment.

ISL has developed nonlinear operators that have the potential for implementation on FPGAs and can be used to carry out correlation estimation, matched filtering, adaptive filtering, and neural networks. The objective of Phase I of this SBIR was to explore the possibility of doing corrrelation estimation in FPGA processors. Is implementation of these estimators feasible in FPGA technology? How do the different estimators compare in terms of performance measures appropriate to an FPGA implementation? How do the estimators compare in a specific application to each other? Can FPGA processors be connected in a parallel configuration to afford more processing power?

In Phase I of this SBIR, we accomplished the following goals set out in the Phase I proposal:

- Chose an FPGA 'style' in which to implement the correlation estimators.
- Encoded the conventional, hybrid sign, and quadruplex estimators in a very high level design language (VHDL) and generated schematics to establish the feasability of VHDL to FPGA implementation.
- Simulated the correlation estimators and determined area/time tradeoffs for correlation estimators for both bit-serial and bit-parallel implementations.
- Considered the relative computational complexities of the different correlation estimators relative to one another.
- Developed a model for and insights into a multiple interconnected FPGA reconfigurable computer and studied potential applications of this reconfigurable computer architecture to applications of interest to BMDO.
- Studied the effect of correlation estimation errors and compared computational complexity in a specific example: Direction of Arrival Estimation.

The correlation estimators we have developed and implemented have a multitude of applications for BMDO. FPGA-based reconfigurable computing architectures using these operators could significantly enhance the capabilities of a real time missile defense system without requiring the size, power, and expense of conventional parallel processing systems. We will discuss some of these capabilities in the following section.

1 Introduction: The Benefits of Reconfigurable Processing for BMDO

1.1 Conventional Processing Limitations

Historically, missile-borne avionics used processing approaches employing a general-purpose microprocessor (GPM) as the computational unit of choice. In recent years, some of the functions requiring extensive localized computing have been committed to Digital Signal Processing (DSP) processors. DSP processors and general-purpose microprocessors differ basically in how they are primarily used. Microprocessors typically aren't used in direct signal path real-time computation and need to shift around their workloads before responding to another user command. Exceptions include some embedded computer architectures with suitable throughput rates for the application of interest. DSP processors are more prevalently geared for processing signals that originate directly from the analog-to-digital signal source and are usually dedicated to computing a sequence of single tasks at a time. However, both *general-purpose microprocessors and DSPs are limited in I/O bandwidth and execute Fixed Instruction Sets.* Although they can be reprogrammed to execute an alternative software program, they *do not have direct hardware reconfigurability and I/O flexibility* that *FPGAs (Field Programmable Gate Arrays) do have.* When combined with FPICs (Field Programmable Interconnects), a very robust architecture can be defined. Alternatively, Application or Algorithm Specific Integrated Circuits (*ASICs*) can have the best in terms of low power, maximum functional density and speed, but they *lack either reprogrammability or reconfigurability.* The benefits of FPGA compared to traditional processors include:

- FPGA implementations are potentially many times faster than DSP.
- FPGA's can handle variable precision computations whereas DSP's handle only either fixed or full floating point.
- ASICS are not re-usable for other tasks, FPGA's are reconfigurable.
- Algorithms and Interconnect can be dynamically and on-the-fly reconfigured on FPGA/FPIC processors which is not the case with fixed GPM or DSP processor boards.
- FPGA/FPIC architectures can implement VLIW (Very Long Instruction Word) computing for wide, concurrent and very fast processing.
- FPGA functional density is rapidly increasing, approaching million+ CLB (Combinatoric Logic Blocks) and can potentially implement any COTS Processor Architecture. This would enable backward compatibility with legacy software already developed and enables the latest in algorithmic technology insertion.

1.2 Flexible Field Programmable Gate Array Implementations

The time to market within the VLSI design cycle has rapidly shrunk at a time when system complexity has intensified considerably. These constraints and other pressures have led developers to seek methods of alleviating this dilemma in order to shorten the design cycle and reduce costs. Expeditious planning and development cycles has evolved into the norm for just remaining competitive in today's complex and booming telecommunications markets. Field programmable gate array (FPGA) technology has been adopted as one of the primary methods of alleviating the situation at hand. Ironically, the very tools that are often dismissed today as simply a by-product within the application specific integrated circuit (ASIC) design cycle may grow to encompass a lion's share of the end product as well. Leading FPGA manufacturers are planning to provide 400,000 gate FPGAs with built-in features such as DRAM before the end of the decade.

The flexibility and quick response of FPGA technology to the changing market scene has led it to be utilized to perform a variety of different tasks. In the process, a natural progression in terms of usage has quietly been going on. At first systems were used primarily for their rapid prototyping capability in industry and system emulation in academia. And this remains largely the case today. However, as more people become aware of the applications of configware this is beginning to change. Furthermore, as partial reconfigurable FPGAs with increasing circuit density become available, then the technology will progress from dynamic reconfiguration to the more demanding on-the-fly reconfiguration necessary for next generation, smart, adaptive and evolvable hardware.

Configware represents a hybridization of the conventional usage of standard central processing units (CPUs) and full custom ASICs in computers. Configware platforms have numerous advantages over traditional forms of hardware and software solutions for processing problems. Innovative hardware platforms, such as a ReConfigurable Processor (RCP), blur the traditional boundaries that exist between hardware and software to bring out the best features of both worlds. A system that is blazingly fast but extremely adaptive to its operating environment. No longer is hardware strictly limited to application specific scenarios. No longer is software just an additive delay, inhibiting system performance. RCP Systems point the way to tomorrow's smart machines, stimulating new advances in hardware and software technology. As a new layer between hardware and software, configware requires a novel programming methodology to create itself. Such technology is referred to as a very high-level description language (VHDL), software which compiles itself into a virtual architecture structure from its binary code. These virtual architectures are then mapped into actual hardware systems to produce the earlier mentioned results.

Field-programmable gate arrays (FPGAs) are integrated circuits which incorporate many identical logic blocks on a chip, are interconnected on the device as needed and can be reused for different applications. In comparison, a custom-designed application-specific integrated circuit (ASIC) is not reusable for other tasks.

The gate-count density has been projected to reach 2 million in the next five years, therefore FPGAs can assume more of the logic implementation on a monolithic processor without requiring a significant amount of area. They can provide parallel data processing, which can render a considerable increase in speed when compared to a dedicated DSP processor solution.

Three-dimensional Inertial Navigation, a significant BMDO application, requires accurate sensor signal processing and an adaptive method for assessing trajectory errors. The sensor signal processing has traditionally been handled by a custom application-specific circuit, which would convert, accumulate and synchronize the sensor-data into a format usable for normal data processing. If a sensor type was modified or replaced, the ASIC would have to be redesigned which requires a significant amount of time in the physical layout of the new device and causes potential schedule delays and increased cost. The FPGA's increasing gate-count density makes this technology a better solution for the sensor signal interface. The speed of data acquisition can be well over 10 MHz, which is at least two magnitudes above the resolution needed for accurate missile navigation and control. The flexibility of the device affords reconfiguration without hardware redesign when a sensor changes its error-model characteristics or is replaced by a different sensor technology.

After the initial signal acquisition, all sensors require compensation for eliminating any errors caused by peculiarities of the device. Coefficients from a bias readjustment up to a third order correction are required to compensate for many non-ideal characteristics. In addition thermal compensation must be included since the output of these devices is extremely sensitive to temperature. These calculations require multiplying the coefficients by the conditioned sensor data, then subtracting each product from the sensor signal. With proper scaling, these compensation coefficients and the subsequent error correction architecture can also be implemented in FPGAs. This technique preserves the parallel processing of the sensor data initiated during the sensor signal processing and therefore provides an increase in computational speed over the typical implementation performed by either a GPP or a DSP. This method also reaps the benefits of selecting the best architecture for implementation, since it's topology is very similar to a low-pass Finite Impulse Response (FIR) filter which has a multitude of possibilities in this technology.

Low-pass FIR filtering that is commonly performed to attenuate undesirable higher frequency errors caused by vehicular vibration and other environmental sources can also be performed in the FPGA. The FPGA filter architecture is much more flexible than a DSP processor implementation since the latter has a fixed multiplier architecture. FPGAs can benefit from using one of several possible architectures. Xilinx Corporation, a major source for FPGA components, performed a study, which compared their standard product to the newest and fastest DSP processor from Texas Instruments (TI). The FPGA implementation using XC4000 series devices resulted in a multiplier up to ten times as fast as the TMS320C6X family (TI) DSP processor.

Multiply-accumulate (MAC) computations are required in many missile detection, navigation and control functions. For instance, the coordinate transformations for converting the sensor orientations from the missile body alignment to the inertial frame coordinates in a strapdown inertial measurement system require several matrix manipulations, which could be accomplished with FPGAs. This implementation would leverage off of the parallel computation possibilities, and combine increased computational speed with improved data synchronization.

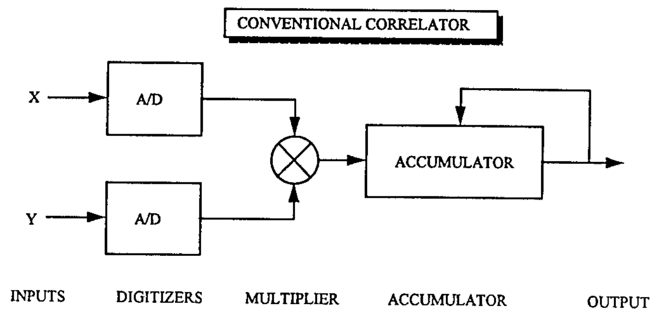
2 Phase I Research Objectives

2.1 Problem Description: Correlation Matrix Estimator (CME)

2.1.1 Case Study: Advantages of FPGA Implementation

In Japan, the Nobeyama radioheliograph monitors the surface of the sun for evidence of solar flares using an array of eighty-four antennas. Generating images of the solar surface in real time requires correlating 3486 distinct pairings of antenna outputs. Since each antenna produces 40 million complex samples per second, computing the correlations consumes just under 560 billion multiply-accumulate operations per second. At 20 MIPS each, nearly 30,000 DSP chips would be required for this task!

The design team, however, found a way to solve the problem with a relatively modest array of 231 custom integrated circuits. The engineers didn't use an exotic technology; the custom ICs were ordinary CMOS devices clocked at 10 MHz. Instead, they hard-limited the data to ±1 as shown in Figure 1 so that all of the multipliers could be replaced by exclusive-OR gates and all of the accumulators could be replaced by counters. This technique is also known as the polarity coincidence method because it counts the number of times the input sign bits agree.



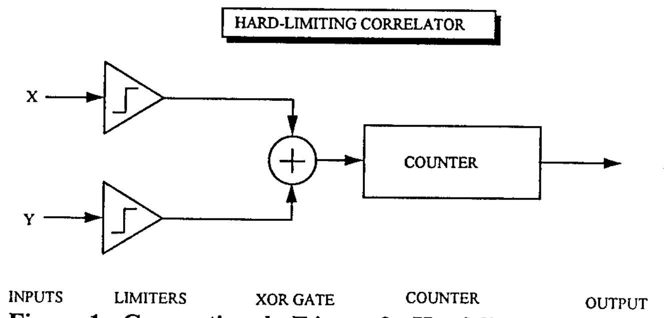


Figure 1. Conventional , F igure 2. Hard-limiting Correlators.

This example illustrates the potential economic benefits of an integrated approach to the design of complex digital signal processing systems. When the algorithm and the hardware are designed jointly and not in isolation, the added flexibility permits design tradeoffs that can reduce system costs by more than an order of magnitude. Research conducted at ISL on the use of nonlinear processing elements for DSP suggests that computationally intensive real-time algorithms can be reformulated in terms of simple arithmetic operations that are easily implemented in digital hardware.

2.1.2 FPGA Based DSP

The advent of the Field Programmable Gate Array (FPGA) provides an opportunity to exploit these techniques to achieve extraordinary processing rates in inexpensive programmable hardware. FPGAs are integrated circuits that provide the functionality of custom logic without the initial cost or development time. Software tools are used to enter and then simulate a hardware design that can then be downloaded into the FPGA (with some FPGAs, this permanently configures the device, but others can be reconfigured with different designs as the need arises). Designs can be entered in schematic form or they can be coded in a hardware description language such as VHDL or Verilog. The design is "compiled" into a configuration file used to program a particular class of FPGAs. Although compiler efficiencies are improving as this technology matures, a little human interaction is usually necessary to meet tough design requirements.

FPGA technology is evolving rapidly; the maximum available density is currently making a transition from 25,000 "equivalent gates" to 100,000-gate devices that are currently available, and 250,000-gate low power (2.5V supply voltage) ones are due 2Q'98. Primitive logic elements such as flip-flops can be operated at clock rates exceeding 100 MHz, but inter-element routing delays generally limit maximum system clock rates to between 30 and 60 MHz. Power consumption depends upon the device, the function being programmed, and the clock rate. Complex, high-speed designs can easily dissipate several watts. 3.3V devices are available for low-power applications, with the goal of 1.8V at .1 micron feature size in the next 5 years.

A typical FPGA is a rectangular array of simple processing elements that communicate across a grid of programmable interconnects. The fundamental characteristic that distinguishes different FPGA architectures is the tradeoff between the complexity of these processing elements and the degree of interconnectivity between elements. A processing element might be as simple as a four-input multiplexer and a flip-flop, or as complicated as four four-input look-up tables and four flip-flops.

FPGAs are not a universal solution to all DSP hardware design problems. Programmable DSP chips are adequate for most applications, and high performance ASICs have been developed for many common functions including FFTs and digital filters. As the example of the Nobeyama radioheliograph shows, however, some real-time applications have computational requirements that far exceed the capabilities of standard components. FPGAs should be considered in these situations if development budgets and schedules are tight or if reconfigurability is important. Array processing and "software radios" come to mind as applications where FPGA-based DSP appears viable.

The key to using FPGAs in DSP applications is to reformulate algorithms in terms of parallel operations that are easily implemented with chains of simple processing elements. The complexity of operations such as addition, subtraction, multiplexing, and comparison are linear in the number of bits used to represent data and are readily implemented in FPGAs. The complexity of a multiplier, on the other hand, is quadratic in the number of bits. The advantages of an FPGA implementation are significantly diminished if the target algorithm uses multipliers.

Since most DSP algorithms rely on multiplication, this would appear to be a severe limitation, but a variety of "shortcuts" are available to avoid implementing multipliers. General-purpose multipliers can be eliminated in digital filters because the input data are multiplied by known constants. Filters with surprisingly good characteristics can be designed with coefficients that are powers of two or sums of a few powers of two. Multiplication by such coefficients can be implemented on an FPGA with a small number of fixed bit-shifts and adds. "Distributed arithmetic" approach replaces multipliers with look-up tables and adders.

2.1.3 Nonlinear Correlation Multipliers

The introductory example illustrates the use of hard limiters to estimate correlations, but there are a number of other nonlinearities that simplify multiplications as shown in Table 1. These nonlinearities are designed to work with zero-mean, equal variance Gaussian data, but useful results are generally obtained from other zero-mean distributions.

Table 1. Covariance Estimators.

Conventional	$\frac{1}{N}\sum_{i=1}^{N} X_i Y_i$
Hybrid Sign	$\frac{\pi}{2}\left(\frac{1}{N}\sum_{i=1}^{N}\lvert Y_i\rvert\right)\left(\frac{1}{N}\sum_{i=1}^{N}\operatorname{sign}(Y_i)X_i\right)$
Quadruplex (See Table 2)	$\frac{\pi}{2}\left(\frac{1}{N}\sum_{i=1}^{N} f(X_i, Y_i)\right)\left(\frac{1}{N}\sum_{i=1}^{N} g(X_i, Y_i)\right)$

Table 2. The quadruplex transformation.

	f(X, Y)	g(X, Y)
$-X \le Y \le X$	Y	X
$-Y \le X \le Y$	X	Y
$-X \ge Y \ge X$	$-Y$	$-X$
$-Y \ge X \ge Y$	$-X$	$-Y$

The hybrid sign estimator from Table 1 can be implemented as a pair of accumulators. An experimental design implemented on a Xilinx 4000 series FPGA used a pair of vendor-supplied macros (a sixteen-bit adder/subtracter and a sixteen-bit register) connected to compute $\sum_{i=1}^{N}\lvert Y_i\rvert$. A slight modification to this arrangement yielded an accumulator design for computing $\sum_{i=1}^{N}\operatorname{sign}(Y_i)X_i$. In operation, both accumulators would be cleared before processing a sequence of N (X, Y) pairs. The accumulator outputs would then be read by a conventional programmable processor that would perform the remaining scaling and (single) multiplication to evaluate the estimator. This processor would use the estimated covariance to perform some aspect of the DSP function.

A resulting implementation would use 18 FPGA processing elements. A subsequent timing analysis indicates that the design would operate at a maximum clock rate of 33.2 MHz. If the object is to estimate a single covariance then the FPGA offers no significant advantages over a conventional DSP chip. On the other hand, if the object is to estimate a symmetric 10x10 covariance matrix, a systolic array of 55 pairs of accumulators operating in parallel could do the job at a rate equivalent to over 1.8 billion multiply-accumulate operations per second. Replicating the hybrid sign arithmetic circuitry 55 times would require fewer than 1000 processing elements, a quantity that could be comfortably fit onto a single high-density FPGA. Conventional programmable DSP chips, with instruction rates well under 100 MHz, are poorly matched to this task.

2.1.4 Performance Tradeoffs

Different nonlinear multipliers offer different tradeoffs between performance and complexity. The more complicated nonlinearities retain more information and thus offer improved performance (lower variance). The improved performance, however, is achieved at a cost of additional hardware. No single technique is "best" for all applications.

The performance of the hybrid sign estimator is compared with that of the quadruplex estimator in Figure 3. The variance of the each estimate is normalized by the variance of the conventional estimator and then shown as a function of correlation. These curves are generated from approximations that are valid for large numbers of Gaussian pairs $\{(X_i, Y_i)\}$ where (X_i, Y_i) is independent of (X_j, Y_j) for $i \neq j$. The hybrid sign estimator provides the greatest computational simplicity, but the reliability of the estimate is relatively poor. Another shortcoming is the variation in performance with correlation; if you need to specify a fixed number of samples to provide a desired variance, without prior knowledge of the correlation you must assume the worst-case performance at zero correlation.

Table 3. Correlation estimators.

Conventional	$\hat{\rho} = \dfrac{\sum_{i=1}^{N} X_i Y_i}{\sqrt{\left(\sum_{i=1}^{N} X_i^2\right)\left(\sum_{i=1}^{N} Y_i^2\right)}}$
Polarity Coincidence	$\hat{\rho} = \sin\left(\dfrac{\pi}{2N}\sum_{i=1}^{N} \operatorname{sign}(X_i)\operatorname{sign}(Y_i)\right)$
Hybrid Sign	$\hat{\rho} = \dfrac{\sum_{i=1}^{N} \operatorname{sign}(Y_i) X_i}{\sum_{i=1}^{N} \lvert Y_i \rvert}$
Quadruplex	$\hat{\rho} = \dfrac{2\left(\sum_{i=1}^{N} f(X_i, Y_i)\right)\left(\sum_{i=1}^{N} g(X_i, Y_i)\right)}{\left(\sum_{i=1}^{N} f(X_i, Y_i)\right)^2 + \left(\sum_{i=1}^{N} g(X_i, Y_i)\right)^2}$

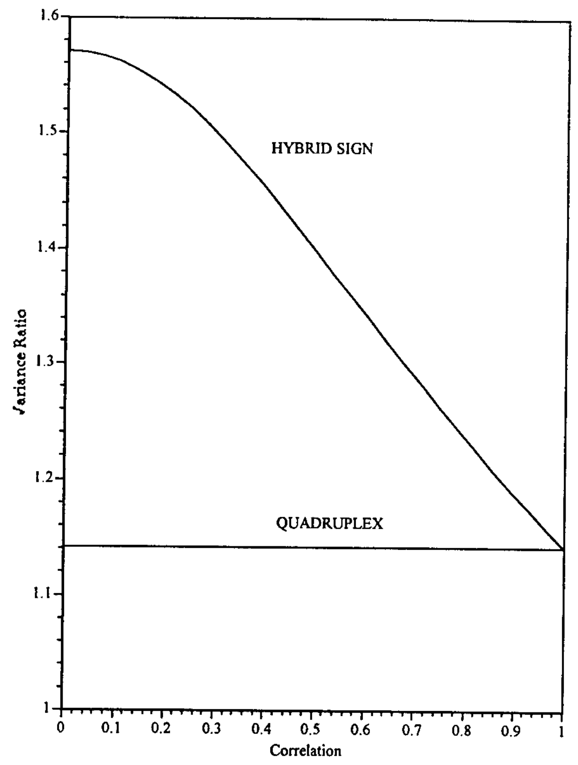


Figure 3. Covariance estimator variance (normalized by the variance of the conventional estimator) for independent Gaussian pairs as a function of correlation ρ.

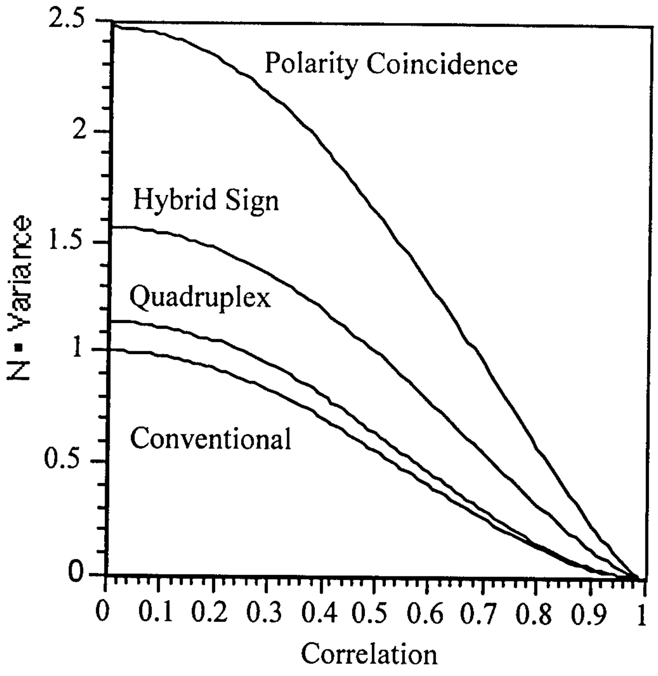


Figure 4. Correlation estimator variance as a function of correlation ρ for independent Gaussian observations.

In some applications the input signal level is a "nuisance parameter" that is often eliminated by a suitable normalization that yields an estimate of the correlation coefficient $\rho \in [-1,1]$. Corresponding to each covariance estimator in Table 1 is an estimator of the correlation coefficient as shown in Table 3, and the differences in performance between these correlation estimators are shown in Figure 4. The tradeoffs between the normalized hybrid sign and quadruplex estimators are similar to those encountered with the corresponding covariance estimators. The polarity coincidence estimator offers extreme simplicity at a cost of greater variance than either of the other nonlinear techniques. The differing tradeoffs between performance and complexity of the four correlation estimators are illustrated in Figure 5. These relative complexities are only representative and actual differences between implementations will vary depending upon the technology employed and the choice of design tradeoffs.

2.1.5 Generalization to Complex Signals

Many digital signal processing algorithms are designed to operate on complex signals resulting from quadrature downconversion. Each of the covariance estimators in Table 1 may be generalized for complex data $Z_i = X_i + jY_i$, $W_i = U_i + jV_i$ as shown in Table 4.

In Figure 6 the variances of the complex hybrid sign and complex quadruplex covariance estimates are normalized by the variance of the conventional estimator and then shown as a function of correlation for three different values of phase. The curves are generated according to approximations that are valid for large numbers of independent, complex, and proper Gaussian pairs. The performance of each of the correlation estimators from Table 5 is shown in Figure 7. The major difference between these figures and the corresponding curves for real Gaussian pairs is that the performances of the hybrid sign and polarity coincidence estimators now exhibit variability with the phase of the correlation as well as with the magnitude.

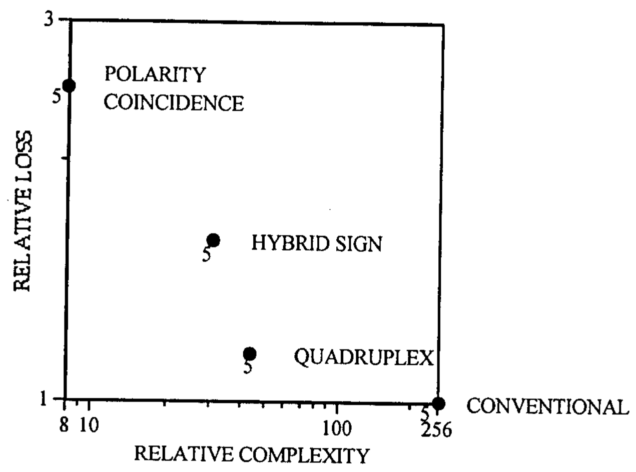


Figure 5. Representative Performance Tradeoffs.

The hybrid sign estimator uses a complex hard-limiter $\mathrm{csign}(\mathrm{W}_i)=\frac{e^{j\pi/4}}{\sqrt{2}}\left(\mathrm{sign}(\mathrm{U}_i)+j\,\mathrm{sign}(\mathrm{V}_i)\right)$ that takes on one of four possible values $\{\pm 1, \pm j\}$. The complex quadruplex estimator requires twice as many additions as the complex hybrid sign estimator, but the improved performance may be worth the extra effort in some applications. Complex generalizations of the correlation estimators in Table 3 are presented in Table 5.

Table 4. Covariance estimators for complex data.

Conventional	$\frac{1}{N}\sum_{i=1}^{N}\mathrm{Z}_i\mathrm{W}_i^*$
Hybrid Sign	$\frac{\pi}{2}\left(\frac{1}{N}\sum_{i=1}^{N}\mathrm{csign}(\mathrm{W}_i)\mathrm{W}_i\right)^*\left(\frac{1}{N}\sum_{i=1}^{N}\mathrm{csign}(\mathrm{W}_i)\mathrm{Z}_i\right)$
Quadruplex	$\frac{\pi}{2}\left(\frac{1}{N}\sum_{i=1}^{N}\mathrm{f}(\mathrm{X}_i,\mathrm{U}_i)+\mathrm{f}(\mathrm{Y}_i,\mathrm{V}_i)\right)\left(\frac{1}{N}\sum_{i=1}^{N}\mathrm{g}(\mathrm{X}_i,\mathrm{Y}_i)+\mathrm{g}(\mathrm{Y}_i,\mathrm{V}_i)\right)$ $+j\frac{\pi}{2}\left(\frac{1}{N}\sum_{i=1}^{N}\mathrm{f}(\mathrm{Y}_i,\mathrm{U}_i)-\mathrm{f}(\mathrm{X}_i,\mathrm{V}_i)\right)\left(\frac{1}{N}\sum_{i=1}^{N}\mathrm{g}(\mathrm{Y}_i,\mathrm{U}_i)+\mathrm{g}(\mathrm{X}_i,\mathrm{V}_i)\right)$

Table 5. Normalized correlation estimators for complex data.

Conventional	$\dfrac{\sum_{i=1}^{N}\mathrm{Z}_i\mathrm{W}_i^*}{\left(\sqrt{\sum_{i=1}^{N}\mathrm{Z}_i\mathrm{Z}_i^*}\right)\left(\sqrt{\sum_{i=1}^{N}W_iW_i^*}\right)}$

Polarity Coincidence	$\sin\left(\frac{\pi}{2N}\mathrm{Re}\left\{\sum_{i=1}^{N}\mathrm{csign}^*(\mathrm{W}_i)\mathrm{csign}(\mathrm{Z}_i)\right\}\right)$ $+j\sin\left(\frac{\pi}{2N}\mathrm{Im}\left\{\sum_{i=1}^{N}\mathrm{csign}^*(\mathrm{W}_i)\mathrm{csign}(\mathrm{Z}_i)\right\}\right)$
Hybrid Sign	$\frac{\sum_{i=1}^{N}\mathrm{csign}^*(\mathrm{W}_i)\mathrm{Z}_i}{\sum_{i=1}^{N}\mathrm{csign}^*(\mathrm{Z}_i)\mathrm{Z}_i}$
Quadruplex	$\frac{2\left(\sum_{i=1}^{N}\mathrm{f}(\mathrm{X}_i,\mathrm{U}_i)+\mathrm{f}(\mathrm{Y}_i,\mathrm{V}_i)\right)\left(\sum_{i=1}^{N}\mathrm{g}(\mathrm{X}_i,\mathrm{Y}_i)+\mathrm{g}(\mathrm{Y}_i,\mathrm{V}_i)\right)}{\left(\sum_{i=1}^{N}\mathrm{f}(\mathrm{X}_i,\mathrm{U}_i)+\mathrm{f}(\mathrm{Y}_i,\mathrm{V}_i)\right)^2+\left(\sum_{i=1}^{N}\mathrm{g}(\mathrm{X}_i,\mathrm{Y}_i)+\mathrm{g}(\mathrm{Y}_i,\mathrm{V}_i)\right)^2}$ $+j\frac{2\left(\sum_{i=1}^{N}\mathrm{f}(\mathrm{Y}_i,\mathrm{U}_i)-\mathrm{f}(\mathrm{X}_i,\mathrm{V}_i)\right)\left(\sum_{i=1}^{N}\mathrm{g}(\mathrm{Y}_i,\mathrm{U}_i)+\mathrm{g}(\mathrm{X}_i,\mathrm{V}_i)\right)}{\left(\sum_{i=1}^{N}\mathrm{f}(\mathrm{Y}_i,\mathrm{U}_i)-\mathrm{f}(\mathrm{X}_i,\mathrm{V}_i)\right)^2+\left(\sum_{i=1}^{N}\mathrm{g}(\mathrm{Y}_i,\mathrm{U}_i)+\mathrm{g}(\mathrm{X}_i,\mathrm{V}_i)\right)^2}$

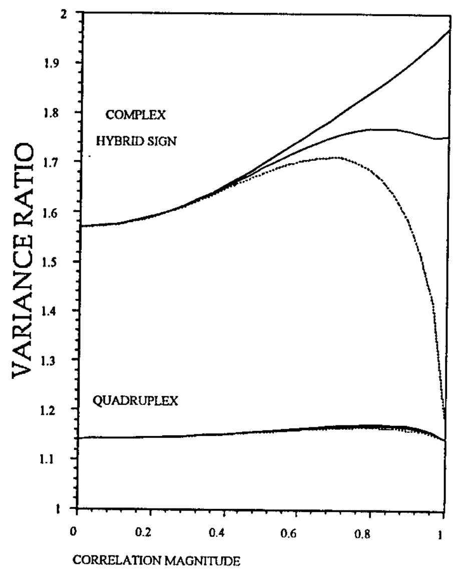


Figure 6: Covariance Variance

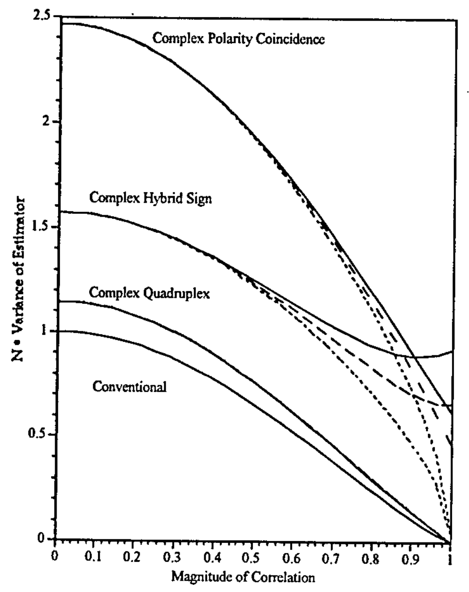


Figure 7: Correlation Variance

Figure 6. Covariance estimator variance (normalized by the variance of the conventional estimator) for independent complex Gaussian pairs as a function of ρ. The magnitude of ρ is given by the horizontal axis and the phase $\boxed{\theta}$ of the correlation is given by the line type. Dotted lines: $\boxed{\theta = 0}$. Long-dashed lines: $\boxed{\theta = \pi/8}$. Short-dashed lines: $\boxed{\theta = \pi/4}$.
Figure 7. Correlation estimator variance for independent complex Gaussian pairs as a function of ρ. The magnitude of ρ is given by the horizontal axis and the phase $\boxed{\theta}$ of the correlation is

given by the line type. Dotted lines: $\boxed{\theta = 0}$. Short dashed lines: $\boxed{\theta = \pi/8}$. Long dashed lines: $\boxed{\theta = \pi/4}$. Solid line: Variance of the conventional estimator.

2.2 Correlation Matrix Estimator (CME) Applications

Estimates of covariance or correlation are seldom obtained for their own sake. More often these estimators are employed as an intermediate stage of processing to obtain some other desired information. Potential applications are abundant and include beamforming, direction-finding, spectral analysis, analysis of cyclostationary signals, adaptive filtering, matched filters, and neural networks. In the sections that follow, computer simulation experiments are used to illustrate the application of nonlinear multipliers to these diverse signal processing topics.

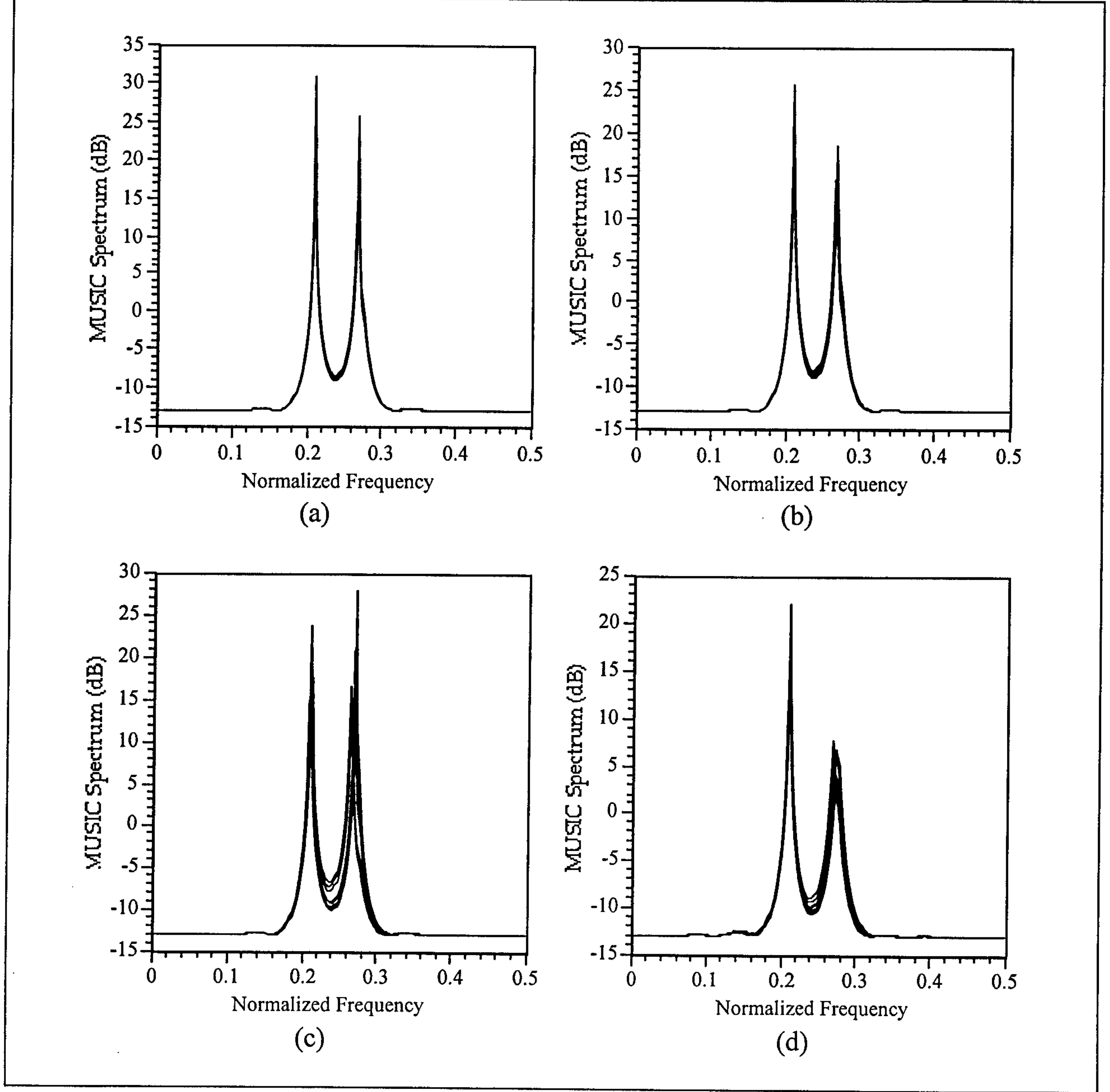


Figure 8. Overlaid results of 10 simulation runs. (a) Conventional MUSIC. (b) Quadruplex MUSIC. (c) Hybrid Sign MUSIC. (d) Polarity Coincidence MUSIC.

2.2.1 Direction Finding

The initial step of many direction finding algorithms is to estimate the covariance between samples collected from different antenna elements. This estimate may be computed with any of the methods given in Table 1. The tradeoffs between performance and complexity will be illustrated with an example using the MUSIC algorithm to estimate angles of arrival of two sources from measurements taken from a uniform linear array. The MUSIC algorithm transforms the covariance function into a function of spatial frequency with peaks near the frequencies of the sources. This function is commonly called a "spatial spectrum" even though the peak heights do not directly correspond to power.

Figure 8 shows how nonlinear correlation estimators affect the output of the MUSIC algorithm. Two sources were generated at normalized frequencies of 0.21 and 0.27 with signal-to-noise ratios of 20 dB and 17 dB, respectively. "Spectra" from ten simulation runs are overlaid in each graph. The conventional correlator produces the graph with the least variation between simulation runs. The most complicated nonlinearity, the quadruplex correlator, exhibits slightly more variation, followed by the hybrid sign and polarity coincidence correlators. The RMS spatial frequency deviations given in Table 6 further demonstrate that the precision of the angle estimate increases with the complexity of the covariance estimator.

Note that the weaker source is much less distinct in the graph generated with the polarity coincidence correlator. If the difference between the amplitudes of the two sources is increased, the peak associated with the weaker source disappears. This limited dynamic range is typical of algorithms based on hard-limiting.

Table 6. RMS spatial frequency deviations of the four algorithms for source at 0.27.

Technique	Deviation
Conventional	$0.54 \bullet 10^{-3}$
Quadruplex	$0.86 \bullet 10^{-3}$
Hybrid Sign	$2.08 \bullet 10^{-3}$
Polarity Coincidence	$2.94 \bullet 10^{-3}$

2.2.2 Stochastic Gradient Filters

In the previous example, an algorithm *explicitly* estimates covariance as an intermediate step in the estimation of the angles of arrival. Many adaptive algorithms *implicitly* estimate covariance. One such algorithm is the well-known LMS or stochastic gradient filter. A diagram of an LMS adaptive filter is shown in Figure 9.

Given an input $\mathbf{x}(n) = [x_1(n), x_2(n), \ldots, x_N(n)]^T$, the problem is to determine the filter weights $\mathbf{h}(n) = [h_1(n), h_2(n), \ldots, h_N(n)]^T$ which minimize the mean squared value of the difference $e(n) = d(n) - y(n)$ between the filter output $y(n) = \mathbf{h}^T(n)\mathbf{x}(n)$ and a reference signal $d(n)$. The solution to this least-squares problem is the set of weights $\mathbf{h}(n)$ that makes the error signal $e(n)$ orthogonal to the inputs $\mathbf{x}(n)$ so that the average value of $e(n)x_i(n)$ is zero.

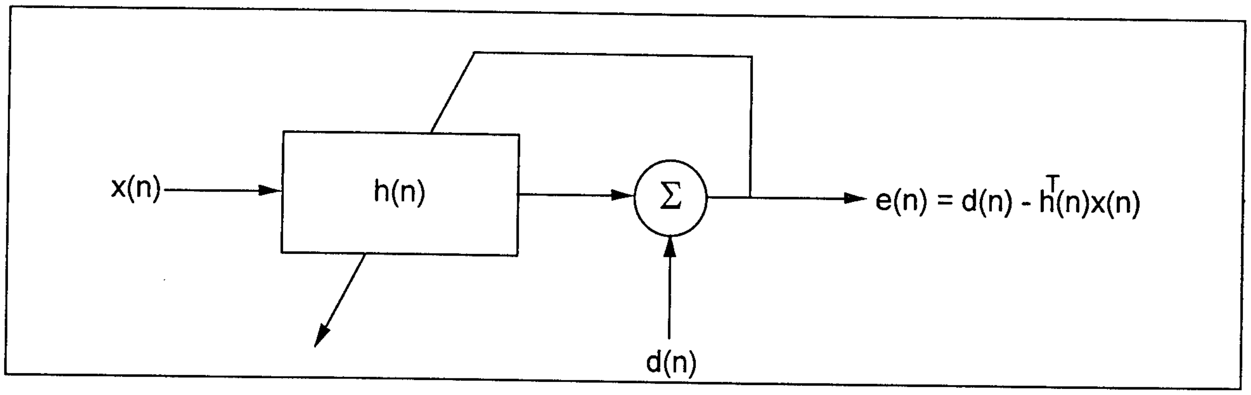


Figure 9. Adaptive filter.

The implicit correlation estimation is revealed by recognizing the update recursion as nothing more than a discrete integrator. As $\mathbf{h}(n)$ approaches the optimal value, the correlation between $e(n)$ and $x_i(n)$ decreases and the average increment in the update recursion approaches zero. This suggests that the update recursion might still work when the multiplication of $e(n)$ and $x_i(n)$ in (1) is replaced by some nonlinear function $f(e(n), x_i(n))$ so long as the average value of the function is zero when $e(n)$ and $x_i(n)$ are uncorrelated. There is no requirement that the average value of the function be *proportional* to this correlation, but odd monotonically increasing functions of correlation would appear to be desirable. Table 7 lists several candidate correlation multipliers taken from the covariance and correlation estimators in Tables 1 and 3 (the signed maximum multiplier is just $f(X, Y)$ from the quadruplex given in Table 2).

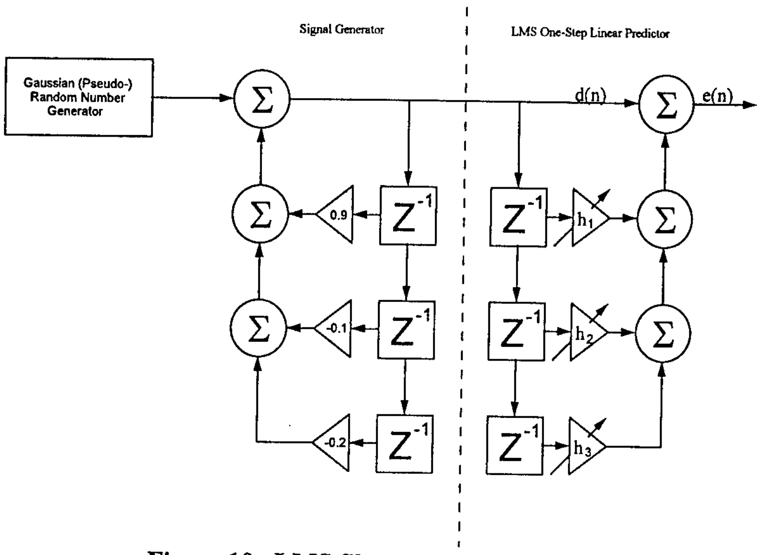

Signal Generator
LMS One-Step Linear Predictor
Gaussian (Pseudo-) Random Number Generator
Σ
0.9
-0.1
-0.2
Z^-1
h1
h2
h3
d(n)
e(n)

Figure 10. LMS filter experiment.

The stochastic gradient algorithm updates the filter weight vector iteratively according to

$$h_i(n+1) = h_i(n) + \mu e(n)x_i(n) \tag{1}$$

where μ is a constant which determines the convergence rate and steady-state error of the filter. Increasing the value of μ provides more rapid convergence until too great a value of μ is reached and the update algorithm becomes unstable. More rapid convergence must be traded off against higher levels of steady-state error caused by increasingly noisy filter weights.

Table 7. Nonlinear correlation multipliers.

Filter Type	$f(e,x_i(n))$
Conventional	$e\,x_i(n)$
Signed Maximum	$\text{sign}(e\,x_i(n))\min(\lvert e\rvert,\lvert x_i(n)\rvert)$
Signed Regressor	$e\,\text{sign}(x_i(n))$
Signed Error	$\text{sign}(e)\,x_i(n)$
Signed Product	$\text{sign}(e\,x_i(n))$

The performance of each of the multipliers in Table 7 was evaluated in a computer experiment as shown in Figure 10. The input vector $\mathbf{x}(n)$ was obtained by taking sequential samples from an autoregressive process generated by passing "white" Gaussian noise through a third-order recursive filter. The reference signal was taken to be the "new" sample of the process so that the adaptive filter was operating as a one-step linear predictor.

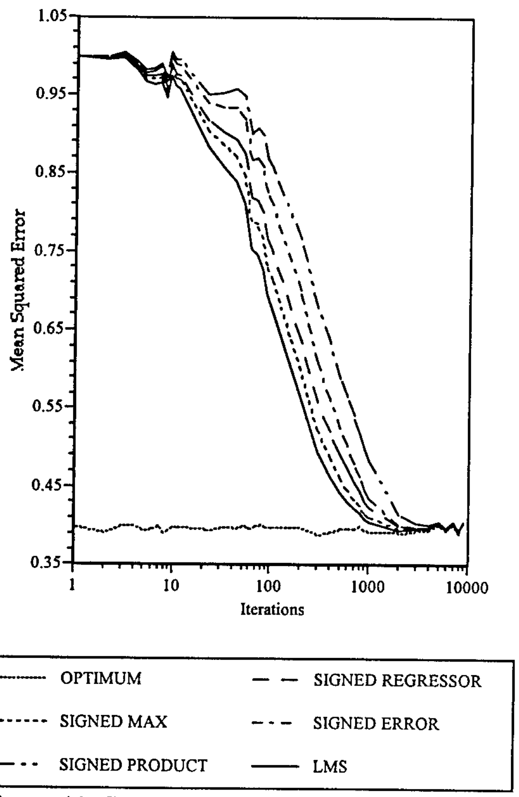


Figure 11. Dynamic behavior of mean squared prediction error.

The evolution of the mean squared error over time is shown in Figure 11 for each of the multipliers in Table 7. Each curve was generated by averaging the results of 10,000 trials. The convergence factor μ was carefully adjusted in each case to provide a common value of steady-state error. This step was necessary because each update provides a different tradeoff between convergence rate and steady-state error for the same value of μ. The results seem to confirm that better performance is available as the multiplier becomes more complicated.

2.2.3 Detection

In all of the examples studied thus far, the conventional multiplier provides the best performance and the performances of the estimators based on nonlinearities are ranked in order of the complexity of the nonlinearity. The other feature common to these examples is the use of Gaussian (or nearly Gaussian) signals. Before any sweeping generalizations are made concerning the relative merits of various correlation multipliers, it might be prudent to run an experiment using something other than Gaussian noise. Consider a very common statistical signal processing problem in communications, the detection of a known signal in additive Gaussian noise. A simple version of the problem is obtained when a single bit of information is conveyed by sending a known signal sequence if the bit is a "1" or sending the same sequence inverted if the bit is a "0". The optimal maximum-likelihood detector for equally likely bit values is the well known matched filter which computes the inner product of the known sequence with the received sequence. The received signal is modeled as

$$x_i = s_i + n_i \tag{2}$$

under H_0 and as

$$x_i = -s_i + n_i \tag{3}$$

under H_1 where $\{n_i\}$ are independent and identically distributed Gaussian random variables with zero mean and variance σ^2 and $\{s_i\}$ is a known signaling sequence. The optimal test for equally likely messages employs the output of a matched filter

$$D = \sum_{i=0}^{N-1} x_i s_i \tag{4}$$

as the test statistic with the decision rule given by: H_0 if $D > 0$ and choosing H_1 otherwise.

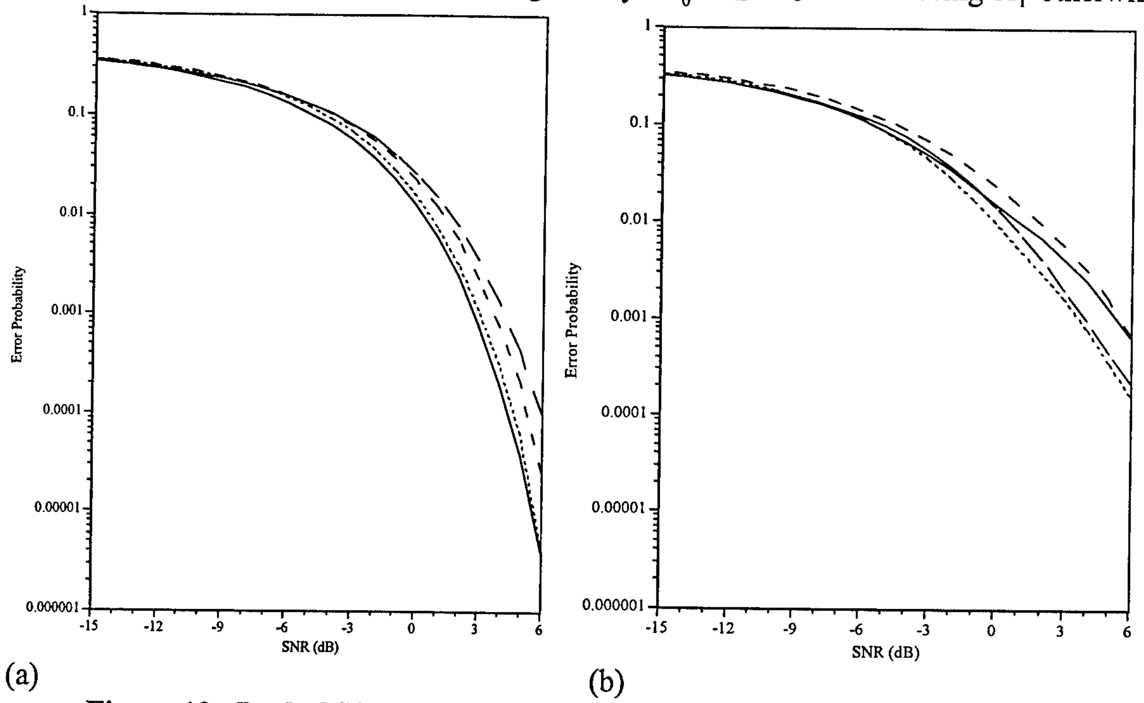


Figure 12. Probability of error in Gaussian (a) and Gaussian mixture (b) noise. Solid Line: Matched Filter. Dotted line: Quadruplex Detector. Dashed line: Sample and Sum Detector. Long Dashed Line: Weighted Partial Decision Detector.

Suboptimal detectors may be designed using the same principles that were applied to the problem of estimating correlations. There are two analogs of the hybrid sign estimator. Processing the received signal with the nonlinearity gives a special case of the weighted partial decision detector statistic

$$D = \sum_{i=0}^{N-1} s_i \operatorname{sign}(x_i). \tag{5}$$

Processing the known signal with the nonlinearity yields the sample-and-sum detector statistic

$$D = \sum_{i=0}^{N-1} \operatorname{sign}(s_i) x_i. \tag{6}$$

The detection analog of the quadruplex correlation estimator is

$$D = \sum_{i=0}^{N-1} f(x_i, s_i) \tag{7}$$

where $f(\bullet,\bullet)$ is the signed maximum function from Table 2.

A computer simulation was used to compare these three suboptimal detectors with the matched filter. A signal sequence of $\{0.25, 0.5, 0.75, 0.5, 0.25\}$ was used to perform 10^6 simulation runs. The estimated probability of error is plotted as a function of signal-to-noise ratio in Figure 12 (a). As expected, the matched filter delivered the best performance closely followed by the quadruplex detector and then the sample-and-sum detector with the worst performance provided by the weighted partial decision detector.

The robustness of each detector was then evaluated by altering the noise model to a zero mean Gaussian mixture with a variance given by σ^2 with a probability of 0.95 and $10\sigma^2$ with a probability of 0.05. A simulation using the same signal sequence and number of trials with the new noise model produced the results shown in Figure 12 (b). The matched filter is no longer optimal for larger signal-to-noise ratios, where the quadruplex detector provides the best performance. The weighted partial decision detector, which provided the worst performance in Gaussian noise, also out-performs the matched filter at the higher signal-to-noise ratios. The sample-and-sum detector provides the worst performance under these conditions.

2.2.4 Neural Networks

The ideas presented in the previous two examples can be combined to show how neural networks might be implemented without multipliers. The simple binary decision problem described by (1) and (2) could be solved by the single-layer perceptron shown in Figure 13 with a suitable training set of received samples $\{x_1(n), x_2(n), x_3(n), x_4(n), x_5(n)\}$ and correct decisions $d(n) \in \{0,1\}$. The weights $\{h_0, h_1, h_2, h_3, h_4, h_5\}$ are updated according to the LMS algorithm with the first weight h_0 corresponding to a bias term that is not paired with an input. The perceptron output $CLASS \in \{0,1\}$ is obtained by comparing the error to a fixed threshold of $1/2$.

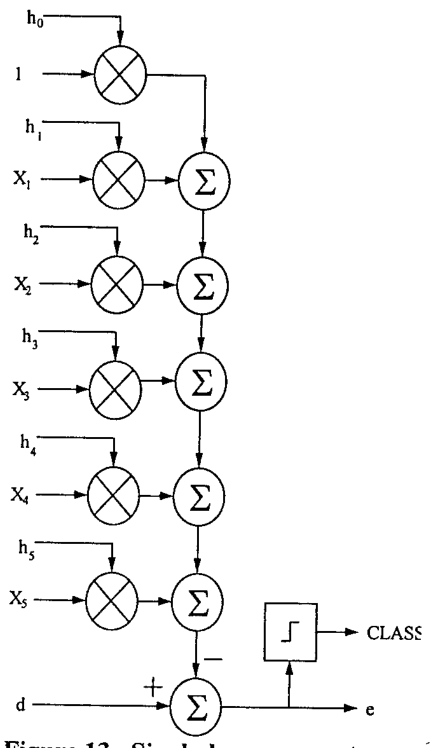


Figure 13. Single-layer perceptron.

Nonlinear functions can replace multipliers in two places, in the weight update and in the computation of the inner product $\sum_{i=1}^{N} h_i(n)x_i(n)$. In Table 8 the conventional approach is compared with a technique using the signed maximum nonlinearity $f(\bullet,\bullet)$ from Table 2. Both algorithms were tested with a computer simulation of a detection problem similar to the one given in the previous section. The noise variance σ^2 was 0.1 and the convergence parameter μ was set to 0.005. The weight vector was initialized to zero and each algorithm was trained using 500 sets of received vectors and correct decisions. The probability of error as a function of time was estimated by averaging 10,000 simulation trials.

The performance of the two algorithms in the simulation experiment are compared in Figure 14. The quadruplex perceptron converges more quickly than the conventional LMS perceptron, but the steady-state error is higher. One important difference between this example and the previous example using the adaptive filter update is that the steady-state errors cannot generally be equalized by using different values for μ. The decision regions of the conventional perceptron are separated by a hyperplane, but the quadruplex perceptron produces a nonlinear decision boundary which may yield better or worse steady-state performance [1,2,3,4,5,6].

Table 8. Conventional LMS and Quadruplex perceptrons.

Conventional LMS	Quadruplex
$y(n) = h_0(n) + \sum_{i=1}^{N} h_i(n)x_i(n)$ $e(n) = d(n) - y(n)$ $h_0(n+1) = h_0(n) + \mu e(n)$ $h_i(n+1) = h_i(n) + \mu e(n)x_i(n),\ i = 1,\ldots,N$	$y(n) = h_0(n) + \sum_{i=1}^{N} f(h_i(n), x_i(n))$ $e(n) = d(n) - y(n)$ $h_0(n+1) = h_0(n) + \mu e(n)$ $h_i(n+1) = h_i(n) + \mu f(e(n), x_i(n)),\ i = 1,\ldots,N$

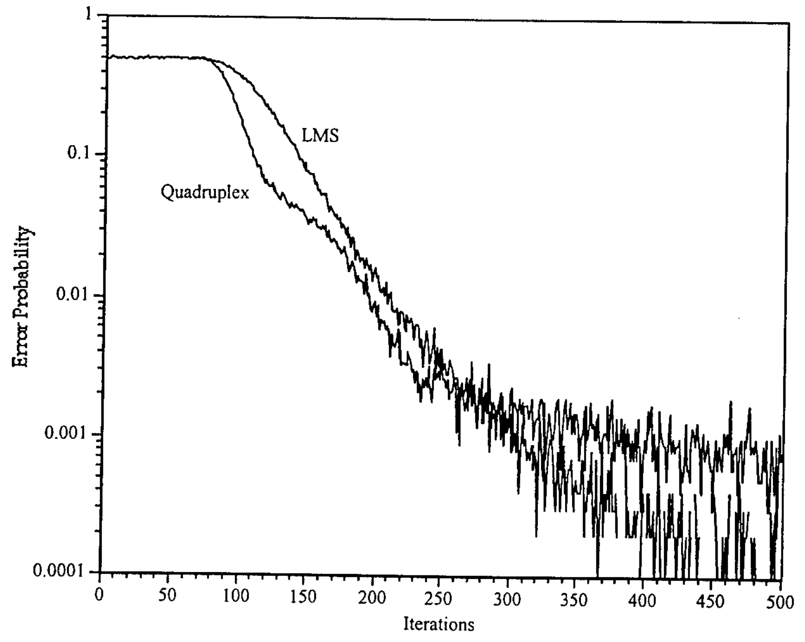


Figure 14. Probability of error trajectory for conv LMS and quadruplex perceptrons.

Multiplier shortcuts allow FPGAs to take on computationally intensive real-time tasks that challenge the capabilities of conventional processor chips. Computing architectures based on FPGAs and nonlinear processing elements can be used to reduce implementation costs with only a small sacrifice of performance. The use of a programmable architecture also reduces development costs and allows the hardware to be reconfigured for different applications. The designs are easily ported to new devices with greater capabilities, so that keeping pace with advances in chip fabrication technology is greatly simplified.

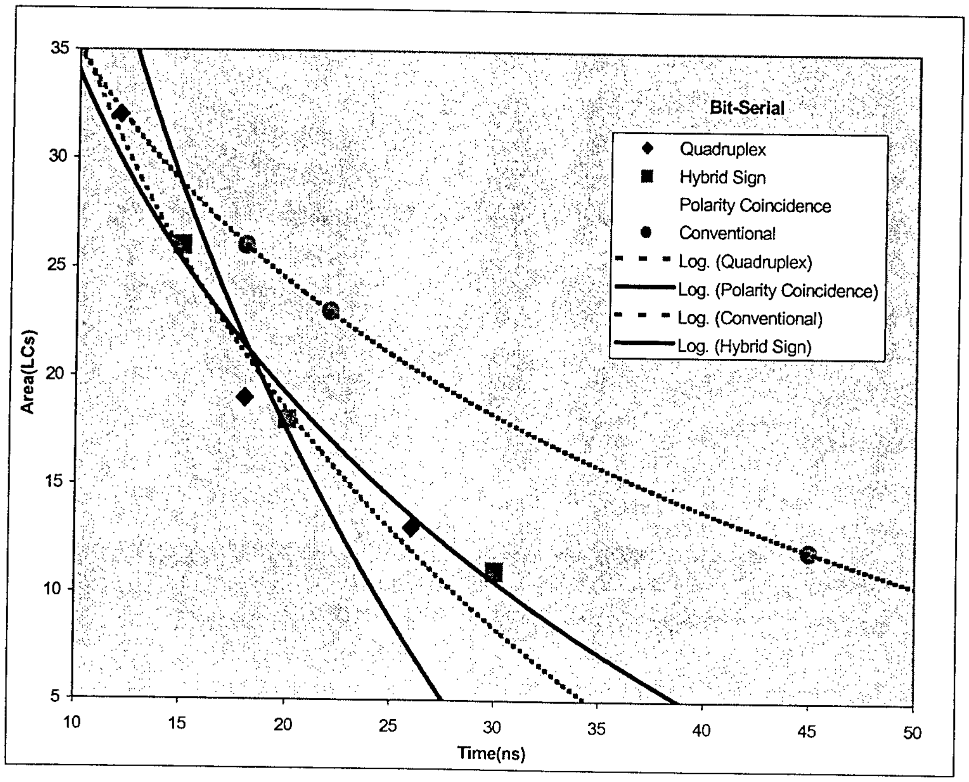
3 Results of Phase I Investigation

3.1 *Task 1: Mapping Basic CME Processing Functions into FPGAs*

3.1.1 Feasibility of VHDL to FPGA Implementation

Configware systems that utilize FPGAs show particular promise for high computational power. Efficient usage of such systems however has remained elusive due to a multitude of factors. Conventional signal processing algorithms do not map well to FPGA resources. This is because their hierarchical gate array structures require specific mapping structures for parallelization and pipelining. VHDL code (see appendix) has been developed for evaluation purposes. The prototype VHDL code was compiled and synthesized via Mentor Graphics using QuickHDL and Exemplar's Leonardo CAD toolkit. Altera MaxPlus2 and Xilinx Foundation Series software was also used for R&D primarily for advanced synthesis and routing placement into specific FPGAs. Other software has been considered and may be used in the future; those by Cadence and Synopsis since they are standard industry CAD vendors. The feasibility of generating FPGA designs automatically from a high-level description of the algorithm has become established in the commercial industry using VHDL and Verilog software code formats. CAD tools by the commercial vendors listed above are in good standing, although it must be noted that they are in a state of constant upgrade due to the plethora of technological breakthroughs. (note: LC = Logic Components, ns = nanosec in Figures 15 and 16).

Figure 15: Bit-Serial FPGA Implementation Costs



3.1.2 VHDL Implementations Comparison of CMEs: Area vs. Time Performance

Cost measurements in terms of logic cells utilized versus the delay time for determining the throughput were taken for serial and parallel implementations. The parallel implementation results and the serial implementation results for the various correlation estimators are shown (in Figs. 15 and 16) per data bit to get an accurate comparison. Obviously, the area consumed per bit has decreased and the throughput rate per bit has increased with parallelization. The results from this study indicate that the parallel implementation of the quadruplex non-linear estimator is optimal.

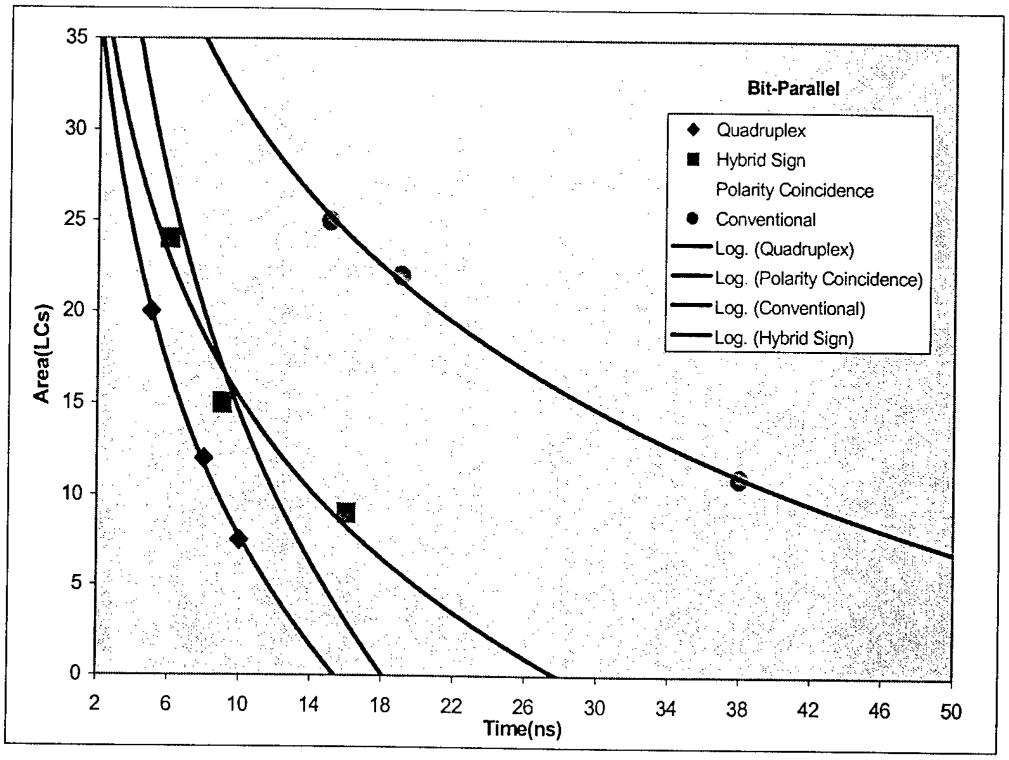


Figure 16: Bit-Parallel Implementation Costs

3.1.3 FPGA Styles

FPGA technology is constantly evolving in accordance with Moore's Law. Virtex is the next generation of Xilinx FPGA technology; a new series of high-performance, high-density, system-level devices with a revolutionary new architecture, built with leading 0.25 micron process technology. The Virtex architecture, with its 2.5-volt supply voltage, offers the industry's first devices capable of directly interfacing beyond CMOS and TTL logic. Voltage supply levels are an intrinsic measure for low-power hybrid and configurable computing. Other interesting developments include the rapid reconfiguration capability of the new Xilinx XC6200 series

FPGA chips, which will enable algorithmic swapping capability for dynamic computation. Conventional DSP processors currently have the advantage for floating point processing, such as the new Texas Instruments C60 processors. These new developments are worth further pursuit. The proposed ReConfigurable Processor architecture resulting from Task 3 will be an enabling technology for BMDO applications, but Phase 2 funding is critical for its development.

3.2 Task 2: Reduced Complexity CMEs Interconnected within Monolithic FPGA

In real-time array signal processing and other advanced signal processing techniques formulating correlation matrices are necessary. The conventional method for estimating the correlation matrix, is to form a sum of products from the received sensor array data. This technique for generating the correlation estimator is computationally expensive for large sample sizes.

In an effort to reduce the complexity, the statistics of the correlation matrix will be investigated. The conventional method involves the expected value of a product $R = E[XY]$. Consider a mapping of the random variables X, Y such that the expected value of the product is a constant times the product of the expected values, $E[XY] = cE[Z]E[W]$. In the case where X and Y are zero mean Gaussian random variables, and using Bussgang's theorem [1] simple transformations can be created with the goal of reducing the computational expense. Using a transformation, Q = h(Y), the correlation between X and Y, R_{XY} can be formulated from the correlation between X and Q, the correlation between Y and Q and the correlation between Y and Y which are R_{QX} R_{QY} R_{YY} respectively [4].

$$R_{XY} = \frac{R_{YY}}{R_{YQ}} R_{XQ}$$

One such simplification is Q = h(y) = sign(Y). The correlation of Y is given by

$$R_{YY} = E[Y^2] = \pi/2 (E\{|Y|\})^2 = (\pi/2) R_{YQ}^2 \Rightarrow R_{YY} / R_{YQ} = (\pi/2) R_{YQ}$$

when Y is zero mean Gaussian. The correlation R_{XY} is now reduced to the product to two, simplistic correlations R_{XQ} and R_{YQ}.

$$R_{XY} = \pi/2 \, R_{YQ} R_{XQ}$$

So the correlation is now of the form $E[XY] = cE[Z]E[W]$ with Z = h(Y) Y = |Y| and W = h(Y) X = sign (Y) X. Over a finite random sample (X_i, Y_i) i = 1, 2 , ...,N the correlation's R_{XQ} and R_{YQ} can be estimated by

$$\hat{R}_{XQ} = 1/N \sum_{i=0}^{N} |Y_i|$$

and

$$\hat{R}_{YQ} = 1/N \sum_{i=0}^{N} X_i sign(Y_i)$$

thus formulating the Hybrid Sign estimator [3].

An alternative approach to estimating the mean of a product is to use a combination of variances fromulating the identity $XY = [(X+Y)^2 - (X-Y)^2]/4$ [2]. If X and Y are zero mean Gaussian random variables then

$$\frac{E\{(X+Y)^2\}}{E^2\{|X+Y|\}} = \frac{E\{(X-Y)^2\}}{E^2\{|X-Y|\}} = \pi/2$$

therefore

$$E\{XY\} = \frac{E\{(X+Y)^2 - E\{(X-Y)^2\}}{4} = \pi/8\, E^2\{|X+Y|\} - E^2\{|X-Y|\}$$

In the formula above E{XY} can be rewritten as the product of to expectations as follows:

$$E\{XY\} = \frac{E\{(X+Y)^2 - E\{(X-Y)^2\}}{4} = cE^2\{|X+Y|\} - E^2\{|X-Y|\} \text{ where } c = \pi/8$$

$$= cE\{|X+Y| + |X-Y|\} * E\{|X+Y| - |X-Y|\} = cE[W]E[Z]$$

Thus the quadruplex transformation is defined as

$Z = [|X+Y| + |X-Y|]/2 = \max(|X|,|Y|)$

and

$W = [|X+Y| - |X-Y|]/2 = sign(XY)\min(|X|,|Y|)$.

This quadruplex transformation can be represented as the following table

	f(X,Y)	g(X,Y)
-X <= Y <= X	Y	X
-Y <= X <= Y	X	Y
-X >= Y >= X	-Y	-X
-Y >= X >= Y	-X	-Y

In the interest of mobile communications complex representation of both of these correlation estimators are necessary. The covariance for a joint distribution for two complex bivariate Gaussian random variables, Z = X+jY and W = U+jV, with means $E[Z] = E[W] = 0$ and variances $E[Z\overline{Z}] = E[W\overline{W}] = \sigma_Z^2$ is

$$R_{ZZ} = E[Z\overline{W}] = R_s + jR_c$$

The complex hybrid sign estimator replaces the sign nonlinearity with a phase nonlinearity:

$$Q_M(re^{i\theta}) = e^{j[\beta + \frac{2\pi}{M}\mathrm{int}(\frac{\theta-\beta}{2\pi/M})]}$$

where M is an integer specifying the quantization in bits and $\beta \in (0,2\pi)$ is a arbitrary phase shift and int(*) is the "nearest integer" operation. By choosing M = 4 and introducing a phase shift of $\beta = \pi/4$ the sign function is now

$$\text{csign}(x+iy) = \frac{e^{j\pi/4}}{\sqrt{2}}\{sign(x) + jsign(y)\}$$

which can take on one of four values in the set {1, -1, i, -i} depending on the real and imaginary parts of Z being positive or negative. To simplify matters the calculation of the product Wcsign(Z) can be formulated in the following table.

$Z = X + iY$, $W = U + iV$	$\mathrm{Re}\{W\mathrm{csign}(Z)\}$	$\mathrm{Im}\{W\mathrm{csign}(Z)\}$
$X >= 0$, $Y >= 0$	U	V
$X <= 0$, $Y >= 0$	-V	U
$X >= 0$, $Y <= 0$	V	-U
$X <= 0$, $Y <= 0$	-U	-V

The complex form for the quadruplex estimator is obtained by using the real form of the quadruplex transformation and appliying it to the real and imaginary parts of the covariance as described below.

$\hat{R}_c = \frac{\pi}{4}\hat{a}_c\hat{b}_c$ and $\hat{R}_s = \frac{\pi}{4}\hat{a}_s\hat{b}_s$

where

$\hat{a}_c = \hat{a}_{XU} + \hat{a}_{YV}$, $\hat{b}_c = \hat{b}_{XU} + \hat{b}_{YV}$, and $\hat{a}_s = \hat{a}_{XV} - \hat{a}_{YU}$, $\hat{b}_s = \hat{b}_{XV} + \hat{b}_{YU}$

and

$\hat{a}_{RS} = \frac{1}{N}\sum_{i=0}^{N} f(R_i, S_i)$ and $\hat{b}_{RS} = \frac{1}{N}\sum_{i=0}^{N} g(R_i, S_i)$

where f, g are the same as in the real quadruplex transformation.

The reduced complexity of the CMEs allows many of these elements to be created and instantiated into a monolithic FPGA structure. They are simply interconnected in a fashion that is referred to as "systolic". This indicates that input and outputs from one element to another flow in a modular, regular and point-to-point communication avoiding broadcasting and a standard bus interconnect protocol. An advantage to this method is that it is naturally implemented in FPGAs and can result in fast pipelined operations.

3.3 Task 3:Multiple Interconnected FPGA ReConfigurable Processor (RCP) Architecture

Configware systems like the proposed RCP (ReConfigurable Processor) described in the next section are fundamentally defined by their ability to adapt or completely reconfigure itself, making them suitable for a whole host of traditional adaptive algorithms and also enabling them to significantly reduce the actual amount of hardware required in an overall system. The potential for small form factor makes configware suitable for embedded systems within today's most advanced systems. Although current generation FPGAs do not have the circuit density nor low power advantages as do custom ASICs for small form factor wireless products, they can still significantly reduce the footprint in the other electronics in UAV (Unmanned Autonomous/Airborne Vehicles) or mobile base or ground stations.

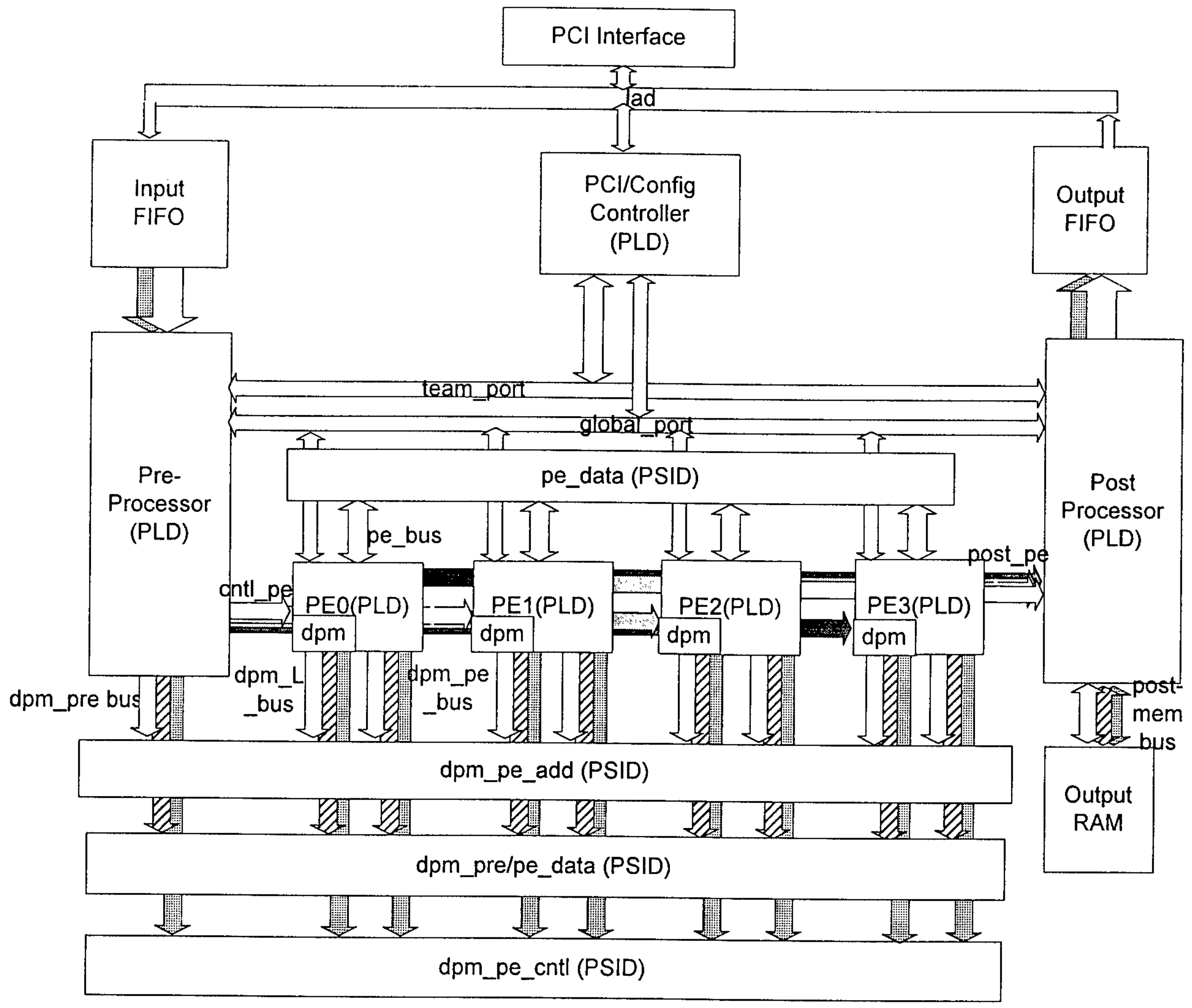
3.3.1 Proposed FPGA RCP Core Architecture Overview

The ReConfigurable Processor (RCP) will be designed as a flexible, high internal bandwidth application specific co-processor that fits into a PC chassis utilizing a Peripheral Chip Interface (PCI). Its primary advantage over other systems is its ability to adapt to do multiple functions efficiently. It is envisioned that the RCP can revolutionize signal processing and data transfer tasks by providing extremely fast and flexible Digital Signal Processing (DSP) functionality for equalization, forward error correction, synchronization, encryption/decryption, compression/decompression, to name a few. However being a configware platform, the RCP is able to move beyond its original purpose. The flexible architecture of the RCP board allows it to be utilized in a variety of applications and it is particularly suitable for any application that demands high computational power. The RCP can be dynamically reprogrammed and reconfigured for a synchronous or asynchronous series of tasks. In addition, the RCP is able to do parallel tasks by loading up multiple algorithms and running them at the same time. For digital signal processing alone, this opens up a whole host of possibilities. These possibilities range from executing multiple DSP tasks on a single channel to switching DSP tasks among a group of heterogeneous channels. In particular, one of the foremost design specifications for the RCP is it can allow adaptive DSP algorithms to run in real-time. The block diagram on the next page shows the architectural design flow of the processor.

The RCP architecture layout consists of 4 FPGA processing elements (PE0->PE3 PLDs) that are connected to each other through direct local buses and through 4 field programmable interconnect ICube IQX160 PSID chips. These processing elements each have their own fast dual port 32k x 16 RAM that can be directly accessed by the PSIDs. This flexibility enables the designer to have multiple degrees of freedom for architecture implementations since one of the problems that designers often face when using other advanced boards are architectural limitations. Architectural limitations have traditionally lead to significantly higher design delays for smart sensors. Two more FPGA (Pre & Post PLDs) processing elements are utilized to take care of pre- and post-processing tasks such as data conversion, pruning and sorting. These processors are connected to the PCI bus through 4k x 18 FIFOs. The post-processor is further connected to a 1 M x 32 single port RAM for data storage of array calculation results and for storing additional data specifically for packet burst communications. Dedicated PCI control logic is used for programming, configuration and PCI interface tasks. A global bus connects all the processors together to a PCI/Configuration Controller device. The RCP is to be designed with multiple easily accessible logic analyzer test headers and LEDs to aid with circuit debugging and for display purposes.

The next few sections give descriptions of some of the algorithms that are suitable for the RCP. The discussion starts with the Viterbi, Reed-Solomon, and Turbo Coding, which are used in communication systems for forward error correction (FEC) signaling; followed by digital beamforming for smart antenna array computing; and then flight telemetry processing. Following the discussions on FEC, digital beamforming and flight telemetry, the discussion delves into advanced hardware topics such as rapid reconfiguration and evolvable hardware for smart sensors. These represent but just a few of the areas where an RCP can utilized to create optimal hardware implementations of algorithms and furthermore, to enable new algorithms to be developed quickly.

RCP SIGNAL BLOCK DIAGRAM



3.3.2 Forward Error Correction (FEC) Signal Coding

Viterbi Decoding

Digital communications will fundamentally transform communication means & multiple system interaction as it takes a ubiquitous role in all electronics. It is acknowledged that the radio frequency (RF) links in the network will be largely asymmetric, with wide bandwidth (video, audio, data) required on the downlink (basestation to remotes), and low bandwidth (audio, keypad) required on the uplink (remotes to basestation). Despite a relatively low bandwidth, the uplink connection presents some difficult technical challenges. FCC, health and safety regulations, and available battery reserves limit the transmitter power levels to less than 600 mW for a handheld device, and practical considerations limit the transmitting antenna size to a small fraction of a meter. Since low signal to noise ratio (SNR) links either impede the achievable data rate or require excess bandwidth, highly efficient usage of the available transmitted signal is a paramount design criterion. Long constraint length (LCL) Viterbi decoders are already used in such applications as deep space communications and voice recognition, however despite the coding gain advantage, they are slow to find widespread commercial acceptance. LCL convolutional coding with Viterbi decoding can significantly improve the reception of digital data. For an uplink implementation, a handheld communication device would encode transmitted data with a simple convolutional encoder and the basestation would receive and decode with the rather more complex Viterbi decoder. The cost of a more complicated decoder present in the smaller population of basestations is expected to be lower than other methods of increasing the received SNR such as larger basestation antennae. Furthermore, as IC technology shows a trend to increasing transistor capacity and lowered costs, this trade-off will continue to favor LCL Viterbi decoding implementations.

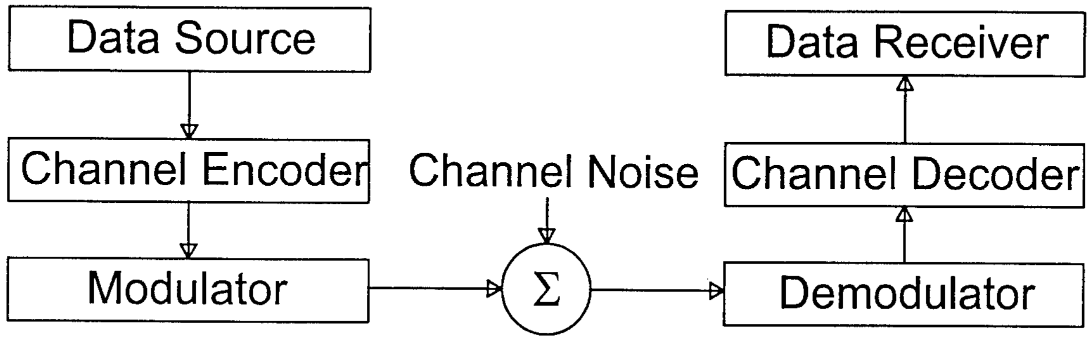


Figure 18. System block diagram.

A typical system, as shown in 18, consists of a data source, channel encoder, modulator, noisy channel, demodulator, channel decoder and receiving system. The channel encoder and decoder are used to correct errors in the information stream caused by channel noise. This is known as Forward Error Correction (FEC). For a given SNR, the *coding gain* is defined as the power level increase (in dB) required by an uncoded channel to match the error performance of a coded channel. Hence coding gain is a measure of the effectiveness of a particular code with larger values an indication of better performing codes. Table 9 shows the asymptotic coding gain of a rate 1/2 convolutional code assuming BPSK with coherent detection. Longer constraint length codes obviously perform better, but require substantially more computation.

Table 9. Coding gain of several good, rate 1/2 convolutional codes.

Constraint Length	Asymptotic Coding Gain (dB)
3	**.97**
7	**3.98**
11	**5.44**
15	**6.53**

Mapping and partitioning are an active area of research in VLSI systems. As presented in [7] [8] [9], *Index Mapping* is a technique to efficiently map a widely used class of algorithms onto a space/time paradigm with immediate representation as the partitioning and scheduling map for the algorithm onto a small, I/O efficient, VLSI array. This class includes the Viterbi Algorithm (VA), the Fast Fourier Transform (FFT), and the Bitonic Sort, among others. A Fixed Shuffle Viterbi Decoder using this technique can be targeted for the ReConfigurable Processor.

Reed-Solomon Implementation in RCP

The code created by Irving Reed and Gustav Solomon in 1960 provides a powerful method for forward error correction of burst errors. In addition, it has an extremely low undetected error rate. This means that the decoder can reliably indicate whether it can make the proper corrections. Reed-Solomon codes have been widely used for error detection and correction for storage media such as compact disks and magnetic hard drives. Other important uses include video data error correction. Unlike convolutional codes, it is a block code and corrects any number of errors in a transmitted symbol. Therefore they have also been implemented as an "outer code" surrounding a Viterbi convolutional error correction design to improve decoder bit error rate performance.

The Voyager spacecraft and Hubble Space Telescope are two examples of space communications systems which have also used Reed-Solomon error correction. For satellite communications, Reed-Solomon codes are frequently incorporated for forward error correction. In fact, the Consultative Committee for Space Data Systems (CCSDS) has standardized a Reed-Solomon format (223 information symbols, 16 symbol correction capability per codeword) which is employed by several satellite systems and the National Aeronautical and Space Administration (NASA). It is the format suitable for the incorporation of the Reed-Solomon algorithm into the proposed RCP.

The Reed-Solomon decoding process is a five-stage process [10] as shown in Figure 19

1. Calculate the syndrome components from the received word. A syndrome is created from a cyclic shifting of the most probable burst pattern.

2. Calculate the error-locator word (polynomial) from the syndrome components.

3. Calculate the error locations from the error-locator numbers which are from the error-locator word.

4. Calculate the error values from the syndrome components and the error-locator numbers.

5. Calculate the decoded code word from the received word, the error locations, and the error values.

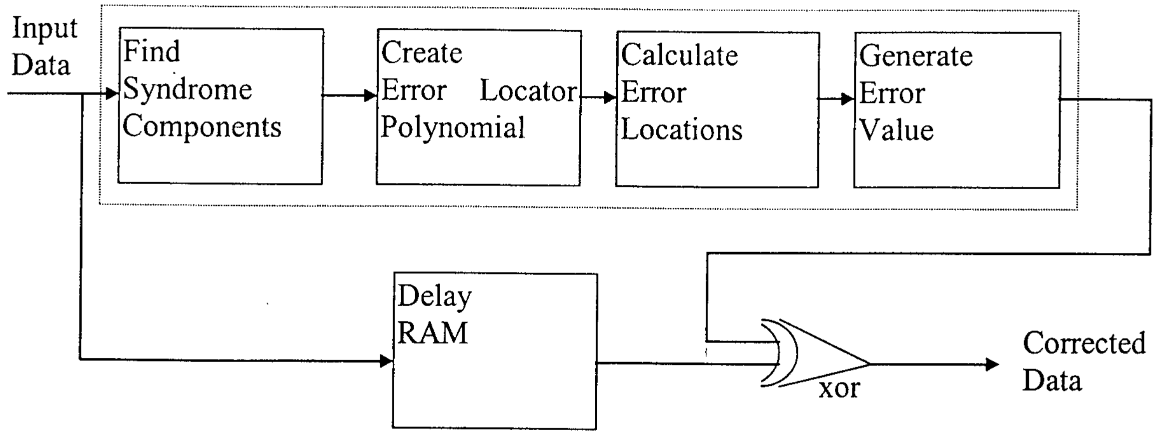


Figure 19. Reed-Solomon decoder block diagram.

Originally the Reed-Solomon decoding algorithm was performed in software due to the computational power required. However, with the advances in VLSI technology, and the development of several architectures which reduced the design complexity, the decoder has also been realized in integrated circuits. As illustrated in the table 10 [11], an FPGA implementation provides a significant speed improvement over a software realization. In addition, the FPGA approach provides a better emulation of a custom integrated circuit design. An FPGA implementation can provide the flexibility to prototype various formats prior to committing to a final architecture.

Table 10. A comparison of architectural implementations.

Implementation	**Throughput**	**Performance/Area Metric (1/(s*M lambda^2))**
Microprocessor (Hypersparc)	46.2 kB/s	.035
DSP Processor (TMS320C30)	18.75 kB/s	.09
FPGA	10 MB/s	20.33
ASIC (LSI Logic)	40 MB/s	406

Turbo Coding for Advanced Communications

Turbo Codes were introduced in 1993 [12] and are considered among the most important developments in coding theory. Researchers around the world have been able to extend the basic idea to other forms of code concatenations, with various applications to transmission over fading channels, band-limited satellite channels, and channels with intersymbol interference. A turbo code is formed by two simple convolutional codes separated by an interleaver. The decoder consists of two Soft-Input Soft-Output (SISO) modules connected by an interleaver and a deinterleaver.

A Turbo encoder is a combination of two simple encoders [13]. The input is a block of K information bits. The two encoders generate parity symbols from two simple recursive convolutional codes, each with a small number of states. The information bits are also sent uncoded. The key innovation of turbo codes is an interleaver P, which permutes the original K information bits before inputting them into the second encoder. The permutation P causes the input sequences for an encoder that produces low-weight codewords to be modified such that the other encoder produces high-weight codewords. Thus, even though the constituent codes are individually weak, the combination can be surprisingly powerful. The resulting code has features similar to a "random" block code with K information bits.

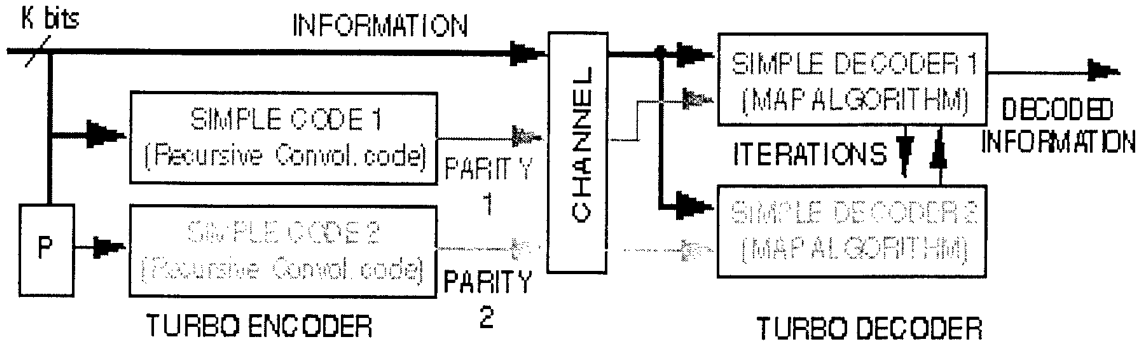


Figure 20. System diagram for a Turbo codec.

Random block codes are known to achieve Shannon-limit performance as K gets large, but at a price of a prohibitively complex decoding algorithm. Turbo codes mimic the good performance of random codes (for large K) using an iterative decoding algorithm based on simple decoders individually matched to the simple constituent codes. Each constituent decoder sends *a posteriori* likelihood estimates of the decoded bits to the other decoder, and uses the corresponding estimates from the other decoder as *a priori* likelihoods. The uncoded information bits (corrupted by the noisy channel) are available to each decoder to initialize the *a priori* likelihoods. The decoders use the "MAP" (Maximum *a Posteriori*) bit-wise decoding algorithm, which requires the same number of states as the well-known Viterbi algorithm. The Turbo decoder iterates between the outputs of the two constituent decoders until reaching satisfactory convergence. The final output is, for this particular algorithm, a hard-quantized version of the likelihood estimates of either of the decoders.

In addition to providing improved performance, 1.0 dB compared to NASA's Cassini concatenated code, turbo decoders are lower in complexity. Turbo codes constructed from two 16-state constituent codes can achieve this signal-to-noise ratio (SNR) performance for 10^{-5} bit error rates (BER) independent of the encoding rate. Turbo decoding time is proportional to the number of states and the number of iterations, unless special-purpose hardware such as the RCP is used to parallel-process some or all of the states. The number of states for turbo decoders is orders of magnitude smaller than the Cassini decoder, and this more than offsets the modest additional amount of iterations required. Interleaver size impacts buffer requirements and decoding delay, but not decoding time. It is the primary determinant of Turbo code performance. Turbo code performance increases as the interleaver size grows.

For latency critical applications such as PCS for two-way vocal communication, the Turbo codes presented by Berrou, Glavieux, and Thitimajshima [12] suffer from a large interleaver. Thus the Turbo decoder is inherently limited (independent of computation resources) because the de-interleaver must wait through a relatively large number of symbols to begin decoding. More recently, the work of Divsalar and Pollara [14], have derived Turbo codes with greatly reduced interleaver sizes at a modest performance penalty. These are far more suitable to low latency applications and represent a good application for the RCP in basestations for Turbo Decoding in PCS.

To implement Turbo Coding on the proposed RCP, several architectural observations need to be made. The encoding and decoding features are distinct. This distinction necessitates different implementations. For real-time two way communication based upon Turbo Coding with minimum footprint requirements, the rapid reconfiguration feature of the RCP can be utilized to switch between implementations. Of the two implementations, the decoder has the most complexity; so it will be discussed here. A Turbo decoder has at its core several small Maximum Likelihood Sequence Estimators (Viterbi decoders) that are cross linked through de-interleavers and run iteratively on the received data sequence. Thus, the most obvious implementation for the RCP is to allocate a single small Viterbi decoder with soft decisions into each PE element. Then, the dual port memories serve to buffer and de-interleave the PEs from each other. The post- and pre-processors are responsible in this situation for all other activity like control and data reconstruction. A more advanced implementation would be to rely upon the rapid reconfiguration capability of the RCP to switch between parallel Viterbi decoding and serial de-interleaving features of Turbo decoders.

Turbo Coding offers significant SNR improvements that are particularly valuable in satellite telemetry systems. The RCP board is intended for satellite telemetry systems that are readily configurable to a wide variety of communication protocols and signal processing requirements, an example application being digital beamforming and blind signal separation.

3.3.3 Digital Beamforming

The recent explosion in cellular and satellite communications has propelled a concurrent effort in the bandwidth expansion of these technologies. An important technique in increasing data throughput in these systems is the use of digital beamforming. Beamforming aims the input beam in a particular direction and attenuates incoming beams from other directions. Beamforming avoids the mechanical wear and tear that a rotating antenna undergoes yet still provides the improved signal reception due to mitigation of unwanted signals from other directions. The fact that digital beamforming is used avoids the drift and calibration difficulties that accompany the use of analog processing. Unfortunately, the possible beam widths are often quite large and may admit signals from several sources beyond those of interest. In this case, further blind signal processing may separate out the users from one another and allow isolation and retrieval of the signals of interest [15]. These methods used in conjunction will improve the bandwidth available for cellular and satellite communications making possible exciting new advances in areas such as vehicular technology. Systems such as On-Star which provide an array of services from ambulance/police alert to navigation assistance rely on an increased bandwidth. The expansion of these services to web browsing and video on demand will compel even greater bandwidth which beamforming and blind signal separation will help to provide.

Array processing is the processing of signals from an array of sensors. These sensors may be anything from microphones to superconducting quantum interference devices. In satellite and cellular communications, we usually consider an array of antennas. Figure 21 shows the antenna array used to do digital beamforming in the Kyoto experiment [16], while Figure 22 shows the array of beams formed.

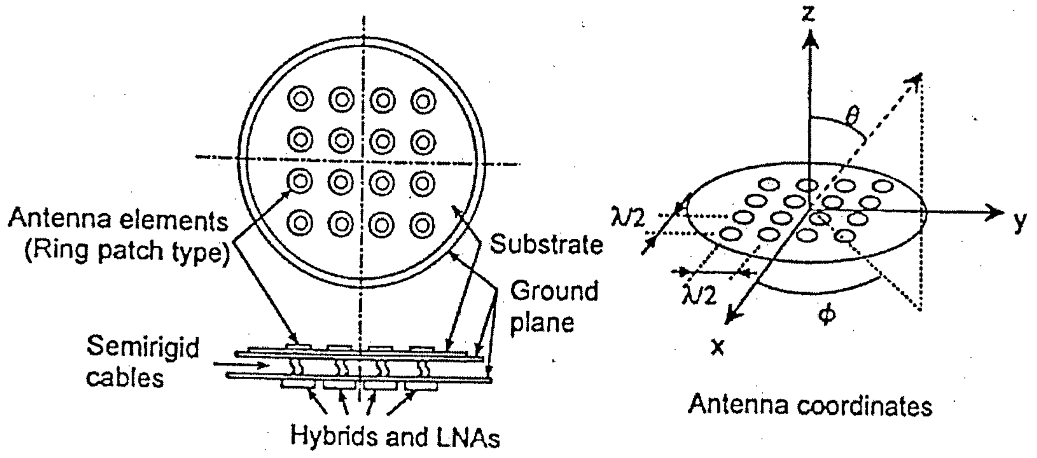


Figure 21. A 4 x 4 antenna array for electronic beam steering.

The objective of beamforming is to delay (although attenuation may also be used) the signal in each antenna so as to constructively sum the signals from a direction of interest while destructively summing the signals from other directions (where the satellite or user of interest is in the direction of interest). In the antenna array space, this amounts to taking the inner product of the input signal with the antenna steering vector in the direction of interest on the antenna array manifold. For the antenna array in the Kyoto experiment, a 2-d FFT was taken of the incoming signal and the resulting signal was filtered to keep the signal of interest and remove unwanted signals. The digital processing stage of the system is shown in Figure 23.

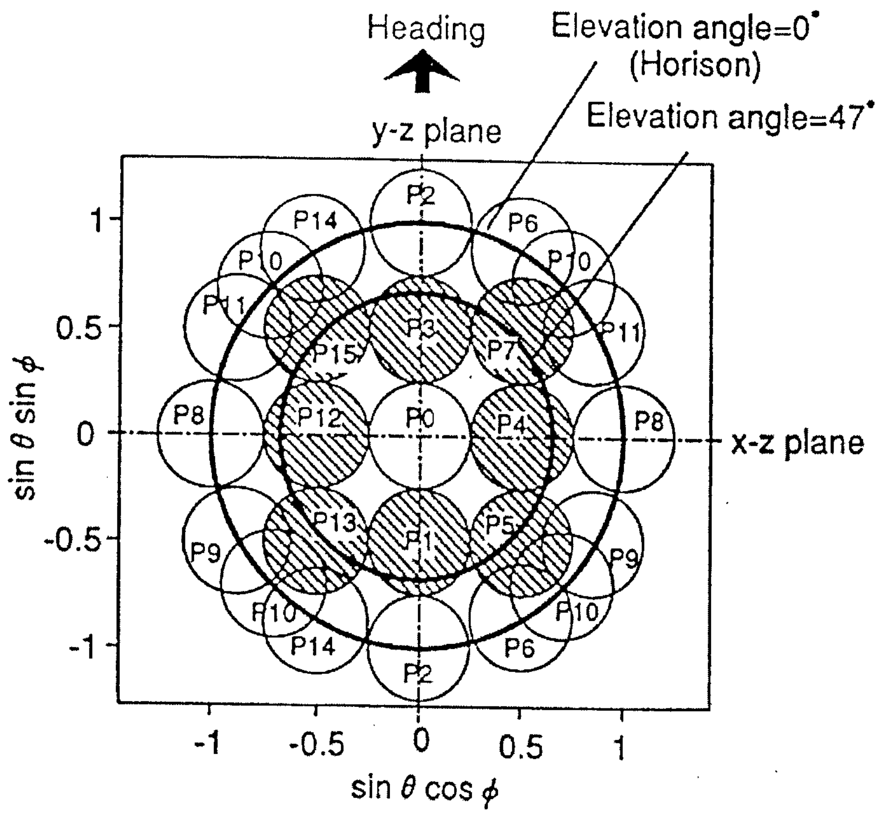


Figure 22. The array of beams formed by the 4 x 4 antenna array.

The core technologies used in this particular experiment have been around for some time. However, how the authors applied the technologies, especially the electronic beam steering to enhance signal reception, is a novel matter for the very important forthcoming generation of SATCOM based communication services. Electronic beamforming and beam steering in the past has been primarily in applications where array was stationary. With the advent of commercial digital SATCOM services (DBS, the Internet, etc.), enhancing reception is paramount, and vehicular technology is a huge potential market for these new SATCOM services. Current wireless technology is quite unreliable in vehicular situations and has, as a result, produced a rapidly growing market that demands higher levels of performance and reliability. The market potentials are significant.

An antenna array, as shown in Figure 24, was mounted on the roof of an RV and driven throughout Kyoto to test the effectiveness of the beamformer in practical situations. In particular, the beamformer was tested under partial and heavy shadowing conditions. The system was able to track the satellite and reacquire the signal quickly under both light and heavy shadowing. Figure 25 shows the performance of the system under shadowing conditions, i.e., as it passes underneath a bridge. The signal level for the array and for a

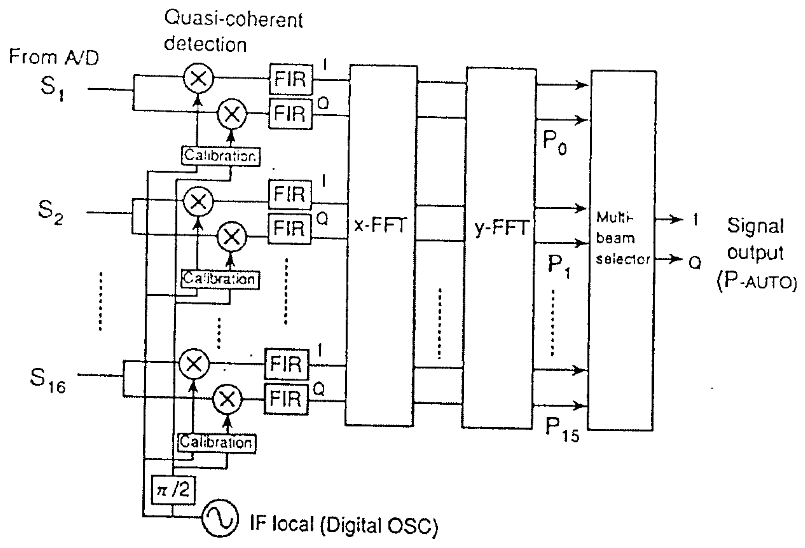


Figure 23. The digital stage of the antenna array processing system.

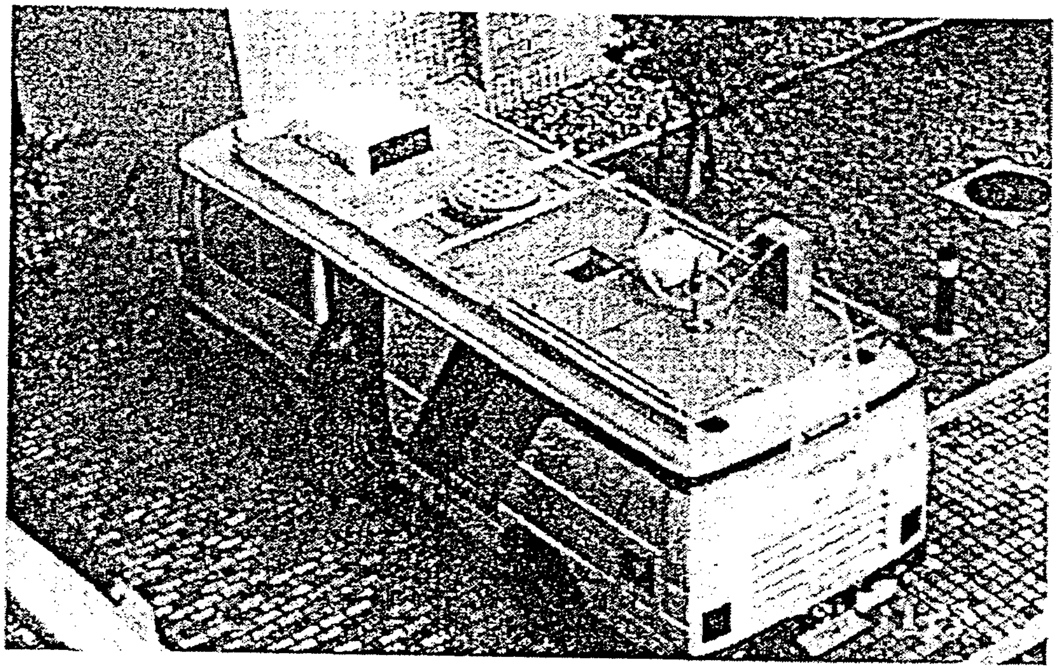


Figure 24. The RV equipped with the roof mounted DBF antenna array.

single antenna are also shown. The antenna array provides approximately 10dB improvement in signal power over a single antenna, so the array provides significant improvement in SNR over a single antenna.

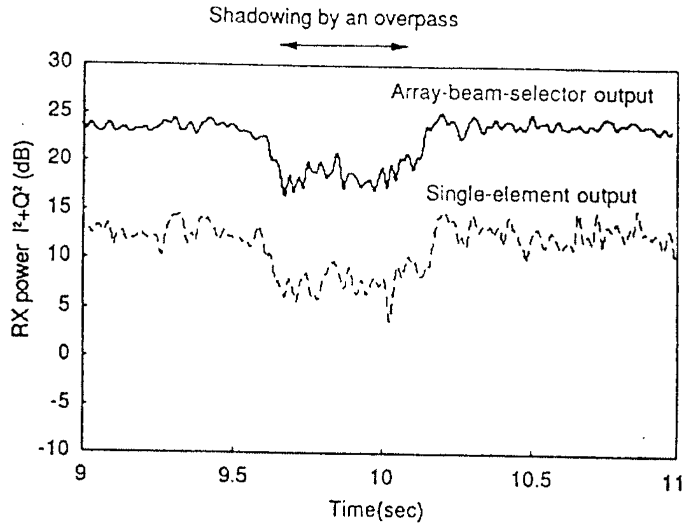


Figure 25. Performance of the antenna array versus a single element under shadowing.

The system was implemented using 10 FPGAs on a printed circuit board (8 implement coherent detection and 2-d FFT, and 2 implement beam selection, control, and interface). The FPGA board is shown in Figure 26

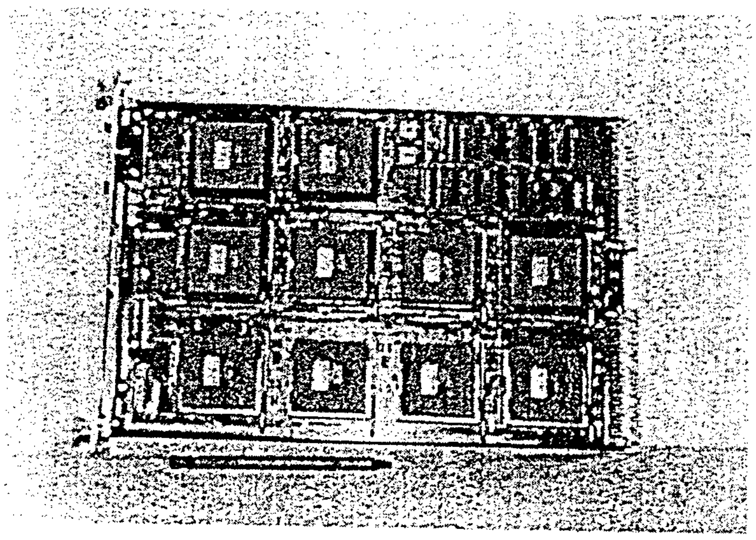


Figure 26. A DSP board with ten Xilinx FPGAs.

Each FPGA chip in the above system had approximately 13,000 gates, so that the entire system consisted of 130,000 gates. However, current FPGA chips have a much higher gate density, and the previous system could be fit on a single FPGA chip. Therefore, if the above system were implemented in state of the art technology such as in the RCP, the capability for implementing more advanced algorithms is realizable due to architectural enhancements.

The beamforming method has some limitations. As shown by the beam pattern in Figure 22, the resolution of the beams is quite large, making beamforming of limited use when sources are close together relative to their distance from the antenna array. In this situation, we propose further signal processing to pull the desired signals out from the composite signal. In particular, we propose to separate the signals using blind signal separation [15]. In this technique the signals from multiple sensors, each of which receives some combination of the signal of interest and the interfering signal, are unmixed using very general properties of the signal. The attractive feature of this technique is that the necessary property holds for most signals. This requires signals from several beams of the beamformer be used. This technique has been shown to separate up to 10 speakers in an audio system where the sensors consist of an array of microphones [17]. This technique could potentially reduce an interfering signal down to a Gaussian noise source in the system, i.e., interference attenuations on the order of 10 dB or more would be possible.

3.3.4 Flight Telemetry Processing

Telemetry is the process of transmitting data measured at a remote location to a distant receiving station for recording and analysis. Real-time recording and analysis involves very high rates of signal processing. The proposed RCP would provide the telemetry community with a low-cost, high-performance integrated configware system for signal processing that fits right into a PCI Mezzanine board slot in a Personal Computer (PC) as an alternative to complex, expensive, and bulky telemetry systems. As a innovative flexible solution, the RCP can be reconfigured at will to do whatever telemetry task or control function is desired [18]. So the RCP can be used for high-speed data I/O processing and storage, including frame synchronization, decommutation, telemetry simulation, FEC decoding and encoding, randomization, reception, bit and frame synchronization, time code processing, NASCOM blocking and deblocking, NASCOM-IP conversion, and high-speed disk logging. Since the RCP is independent of the evolving standards in satellite telemetry used in global communications, the RCP would be able to handle different standards such as TDM, CCSDS and NASCOM. To meet performance needs, the RCP enables high data-rates to be obtained without burdening the host CPU as a co-processor and hardware accelerator.

The RCP could also be a major complement to the current capabilities of typical Visual Telemetry Systems, such as the NeTstar system from L3 Communications, shown in Figure 27. The Loral 550 system could also benefit from the RCP architecture provided that the RCP is mapped to a VME 6U card. The RCP

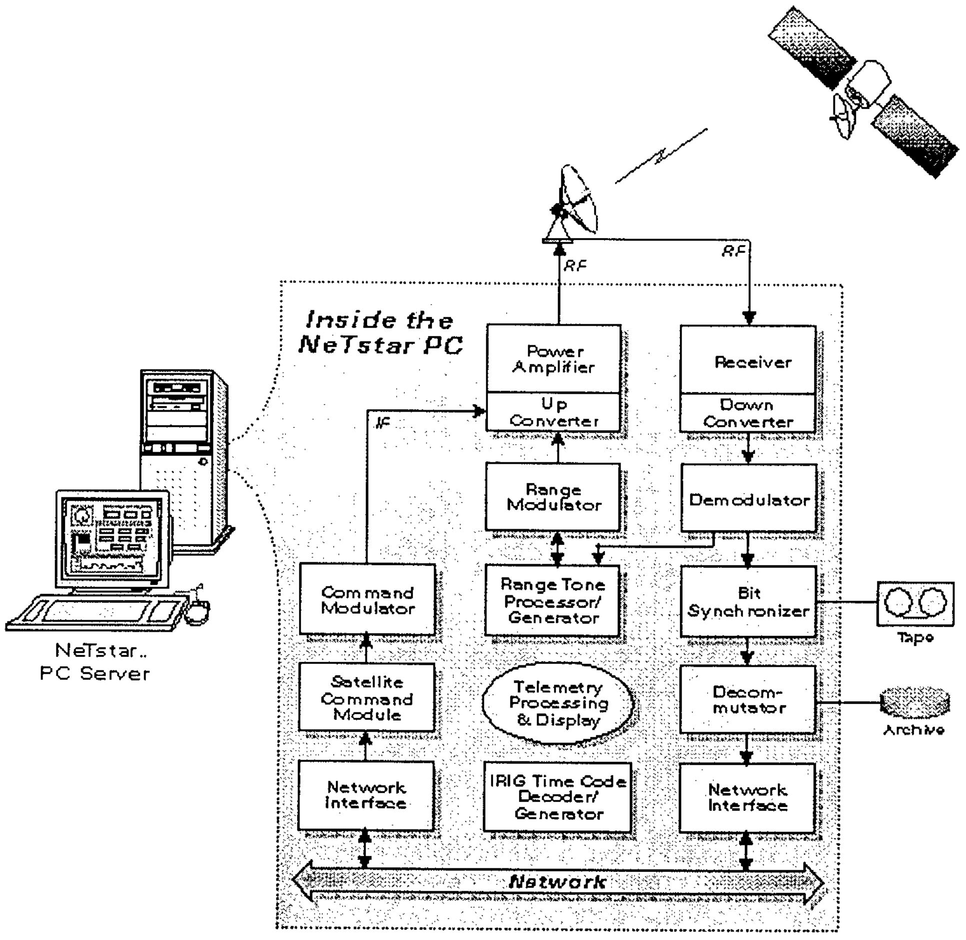


Figure 27. NeTstar block diagram.

will provide the processing solutions needed for advanced telemetry as a hardware accelerator. The only functions that it is not capable of replacing are the RF sections. The RCP can provide a unique answer to telemetry needs. As a configware solution, the RCP will provide significant cost savings for the level of performance that it offers. Most telemetry solutions are software solutions. The RCP is a hardware solution. Most hardware solutions are fixed to a certain handshaking protocol. The RCP can be reconfigured to meet the distinct protocol requirements that exist between different satellite systems. The programmable nature of the RCP gives it a natural ability to input serial and parallel telemetry streams and route them in various ways, making it an ideal solution for front-end telemetry and command systems. Also, the core hardware on the RCP board is not dependent on any specific system requirements. For example, the RCP could alter itself to handle both Ka-Band and L-Band video data processing requirements. So separate specialized processors are no longer necessary since the RCP can mimic their functions. Consequently, the RCP is flexible enough to minimize the need for expensive hardware redesign.

3.3.5 Dynamic "On-the-Fly" Reconfiguration

The RCP can be used as a critical component in systems that require or desire real-time processing through rapid reconfiguration. Real-time processing relies upon computational algorithms whose correctness depends not only on the logical results of computation, but also on the time at which the results are produced. For example, an image processing application for object thinning may demand separate pre-filtering and thresholding steps before [19] or even while running the thinning operation, thus requiring rapid reconfiguration. Implementations of such algorithms can be enhanced by using the RCP architecture. The RCP will enable the user to achieve the utmost reliability due to its flexible architecture and performance through its rapid reconfiguration features. The RCP will utilize state of the art enabling technology and design methodology [20] to achieve these results. Configware in general has been shown to be very suitable for applications that demand fault tolerance. These applications include advanced algorithms that are used in networking, communications and control systems, where rapid reconfiguration provides the best means of adapting when faced with "exogenous shocks"—a term for unforeseen events like the loss of a network router or a malfunctioning machine in an industrial assembly line—which can significantly degrade system reliability and performance. The RCP design has reconfiguration data-flow dependencies in mind and incorporates FIFO structures for tight latency requirements, localized dual port memory for fast computational addressing and deep RAM for quick data storage and retrieval of partial products used in high bandwidth computations. An advanced architectural scheme was created to handle the interconnect between different elements within the RCP architecture. The memory available on the RCP can also be utilized to store different configurations for the FPGA PE PLDs and ICube IQX160s that populate the RCP core and thereby allows the core to not completely be dependent on the PCI bus. These features, and a PCI interface, will allow the RCP to outperform other methods and other reconfigurable processors [21] to do real-time signal processing.

Evolvable Hardware

Engineering can be described as finding and providing solutions to optimization problems within a set of design constraints. Evolvable algorithms are tomorrow's most robust solutions to optimization problems and represent a paradigm shift from today's adaptive algorithms. The RCP architecture with its partial reconfiguration capability is a appropriate platform for exploring evolvable hardware implementations. Evolvable algorithms differ from adaptive algorithms in that they reorganize their operation without *a priori* knowledge of system constraints. The constraints are nominally derived based upon the usage of heuristics or rule-bases like adaptive algorithms. However, they do not remain fixed. The constraints can change according to the tenets of evolution and chaos theory. They often are the most critical element within the architectural implementation of the algorithm, irrespective of whether the algorithm is adaptive or evolutionary. Adaptive architectures have a fixed silicon structure and normally have variable coefficient parameters that are stored either in a lookup table or in memory. Their constraints are often predetermined error terms. An example of this is the weights in the least mean squares (LMS) algorithm. Evolvable architectures on the other hand alter both their basic silicon structure and cell connectivity to operate within a given set of constraints. These evolvable constraints may vary in complexity, implying that they cannot be represented for all known *a priori* conditions. These conditions make up the environment in which the circuit operates. As the conditions change and evolve, the architecture changes to minimize error terms to optimize operations to constraints inferred within the context of the new environment. Evolvable architectures are intelligent in that they encompass a process of natural selection that makes them able to find an optimal solution to any given situation. With situational awareness, they can mutate to retain an optimal solution in real-time systems.

Evolvable hardware has numerous aspects that make it advantageous over other possible methodologies. Architectural changes can be made to optimize tradeoffs between desired performance, area and other constraints based upon actual conditions. For instance, a typical adaptive low pass FIR filter may begin by modifying coefficients to achieve a certain degree of optimization in terms of how sharp its transition bandwidth is. But this may not be sharp enough, so the filter can then begin to evolve until it has twice as many taps. This might give a certain transition band within a relatively short period of time. But suppose the system desires an even sharper cutoff to attenuate a noisy channel. The system could then evolve into a IIR filter to gain an even tighter transition bandwidth at the expense of a slight slowdown in speed.

Evolvable hardware is not too different from today's efficient VLSI design practices. Parallel architecture features allow for quick operation and cellular architecture features that are representative of the divide-and-conquer approach enable incremental structural changes that make partial context switching possible. A key question is what applications require operation in unknown *a priori* constraints? There are numerous applications, some of which we will list here. A primary set of applications include those that utilize genetic algorithms. One possible application is in environments where there are frequent temperature extremes, such as plasma antenna array satellites operating in space. Genetic algorithms implemented in evolvable hardware would enable a expensive satellite to extend their lifespan as well as their effectiveness perhaps by altering itself in real-time based upon solar wind patterns and system usage patterns. Another application is in systems that involve complex network configurations such as those found in the telephone and computer industries. Here a genetic algorithm implementation of simulated annealing might be utilized to provide an optimal solution to the traveling salesman problem for finding the minimal routing delay between two or more points. And predictive analysis for data mining within the stock market could be a rather lucrative application. Rapid changes in stock market indicators used in derivatives can be analyzed ahead of the curve to maximize net profit gains with evolvable hardware. Since advanced signal conditioning based upon vector fields could be accelerated to deal with matrices without full rank or matrices that have full rank but have eigenvalues that are ill conditioned. Other areas of interest include unmanned vehicles and diagnostic applications in medicine. In short, with evolvable hardware, real-time asset reconfiguration enables next generation solutions by enabling smart machine designs. Perhaps someday a doctor could apply a skin patch on a sick patient to monitor the insulin and oxygen levels found in their blood and have it later evolve to monitor cholesterol as the patient begins to recover using the same sensor.

A Future FPGA Technology Niche

FPGA technology brings some unique advantages that are absent from traditional VLSI architectures. The key advantage offered is the ability to physically change through reconfiguration. This allows for generic processing that used to only be possible in software to be sped up by mapping it directly into hardware. This hybrid technology combines the versatility of a standard CPU with the performance of a standard cell custom ASIC. The other key advantage is the shrinking form factor size requirements that reconfiguration enables. The same piece of hardware could be re-utilized to perform a completely different function. For example, a digital handheld real-time video telecommunications application is a prime target as a future application since multiple functions can be realized using the same hardware. Such an application might rely on a few evolvable system configurations to adjust to bandwidth and power tradeoffs. Of course requiring the end-user to have the technical background needed to create their own configuration files using a hardware description language would unduly restrict market opportunities. So it is envisioned that users could download new configurations for a business communications supplier through a modem configuration, allowing the device to be truly multi-functional and enabling a large potential consumer base to exist. Future internal device characteristics of FPGAs are expected to radically change as well. So the current existence of SRAM based FPGA technology should not be thought of as a prelude to the demise of integrated circuit digital design. Ultra high speed interconnects and resulting wire crosstalk still remain a major research challenge that needs to be explored and conquered on the device and systems level. Novel architectures are constantly being introduced and many discrete components have not yet been completely integrated. For example, new Flash technology FPGAs that have half the relative capacitance and seven times the density of their SRAM counterparts with fast and frequent access time requirements are currently under development. Other future developments will probably include, but not be limited to, a merging of sensor arrays with configware for situation awareness. FPGA technology is the only available technology that has the flexibility and performance to make evolvable hardware feasible. As its name implies, evolvable hardware may someday exist with the capability of recreating the complex genetic DNA interactions found only in biologically living creatures.

Today, reconfigurable hardware solutions are expanding to wide variety of engineering applications by leveraging off the increasing density available in FPGA technology and the flexibility of ASIC Field Programmable InterConnects (FPIC). Having hardware commonality between subsystems reduces costly development efforts by re-using the same hardware for different applications. Reconfigurability further accelerates productivity, providing rapid conversion from concept to reality by allowing full system implementations without additional hardware development costs. The motivation for the RCP research was to investigate architectures for the next generation of cost effective, high performance BMDO applications. Given the tremendous diversified growth in telemetry, remote sensing, telephony, and telecomputing, robust processing of multi-source acquired data signals for digital communication through reconfiguration becomes a paramount system concern. Further work in reconfigurable parallel and distributed signal processing architectures will be investigated in Phase II to enable new and improved algorithms to be mapped into the ReConfigurable Processor (RCP). The RCP will be designed, developed, and fabricated during the 2nd phase of this ReConfigurable Processing research project. Such configware solutions are needed to handle the diverse bandwidths, high data rates, varying signal qualities, and variable signal processing needs in current and forthcoming BMDO systems

3.4 Task 4: Correlation Matrix Estimate Comparison in DOA Finding

3.4.1 Introduction

In order to study the effect of error in the correlation matrix estimate in a specific example, we consider direction of arrival (DOA) estimation with the correlation estimators described above. We compared the correlation estimators in estimating the DOA of a sinusoidal signal on an antenna array.
In the conventional, complex hybrid sign and the quadruplex correlation estimates, the variances have been determined theoretically. The conventional variance is:

$$\mathrm{var}(R_{ij}) = \sigma^4 / N$$

The variance for the quadruplex estimate is given by:

$$\mathrm{var}(R_{ij}) = (\pi - 2)\frac{\sigma^4}{N}$$

It has been shown in [22] that both the complex hybrid sign and the quadruplex approximation exibit higher variances than the conventional method. Numerically, these varainces are between 50 – 100% and 14% greater then that of the conventional estimator for hybrid sign and quadruplex respectively.

We applied the correlation estimators to the problem of DOA estimation using Beamforming [23]. Beamforming takes the inner product of the output of an antenna array with a weight vector to produce an output signal (on which further temporal or spatial processing may be done). This inner product is typically used to isolate the signal coming from a particular direction. Assuming there is a signal, which is significantly larger than the background noise impinging on an array:

$$x = As + n$$

where x is the received signal, A is the antenna steering matrix, s is the transmitted signal, and n is additive white Gaussian noise. We can train the weight vector to only allow that signal to pass through the inner product, essentially carrying out a spatial filter of the incoming signal. This is done by using gradient ascent to maximize the power of the inner product output:

$$\max(P(w)) = \max(E(w^H xx^H w)) = \max(w^H R_{xx} w)$$

$$\frac{\partial P(w)}{\partial w} = w^H R_{xx}$$

Furthermore, by examining the weight vector (taking its inner product with signals impinging from all possible directions), we can estimate the Direction of Arrival (DOA) of the signal of interest. This DOA estimate depends critically on the estimate of the correlation matrix R_{xx}, so we will study the effect of the correlation estimate on the DOA estimate.

3.4.2 Computational Complexity

The next issue is computational complexity of the three different correlation estimators. The complexity will be based on multiplications/divisions, additions/subtractions, comparisons, and shift for negative signs. Given an L sensor element array and a sequence of N time samples, the computational burden for $(R)_{ij}$ the ith and jth element in the signal correlation matrix are in the Tables 11 & 12 below.

The comparison of the real conventional, hybrid sign and quadruplex :

Table 11: The complexity for a single element in the correlation matrix

		Multiplies/ Total	Additions /Total	Compares/ Totals	Shift (negative sign)/Totals at most
Real Data	conventional	N	(N-1)	0	0
	hybrid sign	1	2(N-1)	1	2
	quadruplex	1	2(N-1)	8	2
Complex Data	conventional	4N	2(N-1)	0	0
	Hybrid sign	4	4(N-1)	2/	2
	quadruplex	4	4(N-1)	8/	2

Table 12: The complexity for calculating the entire correlation matrix

		Multiplies/ Total	Additions /Total	Compares/ Totals	Shift (negative sign)/Totals at most
Real Data	conventional	NL^2	$(N-1) L^2$	0	0
	hybrid sign	L^2+2	$2(N-1) L^2$	L^2	L^2
	quadruplex	L^2+2	$2(N-1) L^2$	L^2	L^2
Complex Data	conventional	$4N L^2$	$2(N-1) L^2$	0	0
	Hybrid sign	$4 L^2+2$	$4(N-1) L^2$	L^2	L^2
	quadruplex	$4 L^2+2$	$4(N-1) L^2$	L^2	L^2

We also compare the computational complexity of correlation estimators for a specific example, to illustrate, in real numbers, the potential computational efficiency of using the alternative correlation estimator, and especially the quadruplex estimator. The results are tabulated below in Table 13.

Table 13: Calculating the correlation estimates of the different methods

		Total bit level operations	Bit level Multiplies	Bit level Additions
Complex Data	conventional	989,900	960,000	29,900
N = 300	hybrid sign	122,816	3,216	119,600
	quadruplex	371,000	11,200	359,800
Complex Data	conventional	10,199,800	9,600,000	599,800
N = 3000	Hybrid sign	1,201916	3,216	1,199,600
	quadruplex	3,611,000	11,200	3,599,800

3.4.3 Simulation Results

We simulated the beamforming algorithm and the three correlation estimates in Matlab. A sinusoidal signal was assumed to be transmitted from $\pi/8$ degrees relative to the normal to the array. We used a 10 element linear antenna array uniformly spaced at $d=\lambda/2$ where λ is the wavelength of the incoming signal. In our case, we used a 2MHz bandwidth signal sampled at 2MHz to avoid aliasing. The incoming signal was complex as was the antenna array steering matrix and the white Gaussian noise. We assumed no fading and additive white Gaussian noise of varying power. The weight training procedure consisted of updating the weight vector 50 times for each set of time samples (i.e. for each correlation matrix) according to the gradient ascent equations mentioned above. This number of updates was chosen large enough to ensure convergence. We did not address questions of convergence speed in this report, only approximation accuracy. Each set of parameters was run for 10 trials to determine mean and variance of the DOA estimate.

We were interested in the correlation approximations as a function of several variables. We varied the SNR, number of time samples per correlation matrix formulation, and of course, the approximation method (conventional, hybrid, or quadruplex). The variances for the different approximation methods given by Sullivan [4] were predicated on the input signal being independent and identically distributed Gaussian random variables. We therefore used very low SNR to see if our quadruplex and hybrid variances matched up with theory. However, in most practical situations, the signal is larger than the noise. To model realistic situations, we allowed much higher SNR. We were also interested in the tracking ability of these algorithms, i.e. how well would they perform if the signal of interest were moving (as happens in a cellular telephone environment, for instance). This implies that samples must be taken over short time windows, such that the signal has roughly stationary parameters over the set of samples. We therefore varied the number of samples per correlation matrix to see how the correlation approximation biases and variances were affected by number of samples.

The simulation results show good correlation with theory at low SNR and show very interesting behavior at high SNR. The variances for the quadruplex and hybrid approximations, derived assuming Gaussian distributed random variables, showed a 14%

increase from conventional to quadruplex estimates. The biases of our estimates are shown in the plots in figure 17. For low SNR (0, 6, 10), the quadruplex variances are approximately 16% larger than the conventional method variances. This agrees well with the theoretical value of 14%. This indicates that our quadruplex and conventional correlation calculations are working properly.

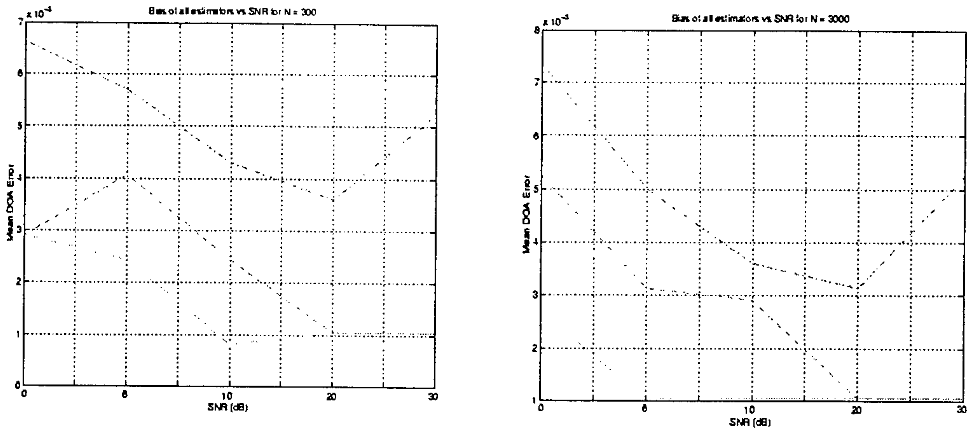


Figure 17: Bias in radians vs. SNR for hybrid sign (dot-dashed line), quadruplex (dashed line), and conventional (dotted line) correlation approximations. N=300 samples on the right and N=3000 samples on the left.

However, a very interesting phenomenon occurs at high SNR (>10dB). The bias and the variance of the quadruplex estimator is actually lower than that of the conventional estimator.

3.4.4 Discussion

Given the above performance results, we can classify the suitability of the various correlation approximation methods under different conditions. The hybrid approximation clearly performs poorly across all SNR levels having a bias of from 25% to 400% larger than the conventional approximation. The quadruplex estimator, on the other hand, performs very well compared to the conventional approximation across all SNR levels. The most striking feature is that the quadruplex approximation has lower bias and variance than the conventional approximation for high SNR. This is unexpected since the conventional approximation produces a theoretically optimal estimate of the correlation matrix. This improved bias and variance warrant further investigation.

These results also impact calculation of the inner product in neural networks. The quadruplex and hybrid sign are simply approximations of inner products [3]. They are therefore applicable to neural networks where the inner product is calculated between the input vector or output vector from the previous layer and the weight vector corresponding to each processing element. Our results suggest that the quadruplex approximation would be most successful in neural network application.

4 Phase I Summary and Recommendations for Phase II

4.1 Summary of Phase I Effort

The Phase I accomplishments can be listed as follows:

- Correlation matrix estimators implemented & verified on an FPGA processor.
- Nonlinear correlation estimators provide up to twice the throughput rate in bit serial and 3-5 times the throughput rate in bit parallel FPGA implementations compared to a conventional estimator
- Nonlinear Correlation estimators require 1/3 to 1/10th the number of bit level operations that the conventional estimator requires.
- Area and time can be traded off against one another to achieve a desired throughput rate given a certain number of FPGA processors.
- The quadruplex estimator provides the least error in estimate while the hybrid sign and polarity coincidence provide the greatest speed increase.
- A Multiply Interconnected FPGA ReConfigurable Processor Core Architecture was specified with applicability to a variety of signal processing functions.

4.2 Phase II Recommendations for FPGA RCP Development Targeted to BMDO Programs

Given the different domains of applicability of FPGA processors and DSP's, we will propose (in phase II) development of a complete hybrid reconfigurable system combining the best of both FPGAs (consisting of the Multiple Interconnected FPGA based RCP core proposed as a result of the Phase I study) and DSP processors. We will then, under close consultation with BMDO, implement an application of interest to BMDO on the hybrid architecture.

The Phase III task will be the field testing of the hybrid architecture on a specific program relevant to BMDO's mission, along with the commercialization of the proposed RCP for other applications.

4.3 Some Potential CME Applications of Particular Interest to BMDO

We complete this report by listing two very important applications of the proposed FPGA technology:

- Adaptive (Kalman) filter tracking of Missiles subject to false target, decoy, and chaff countermeasures.
- Automatic Target Recognition (ATR) of incoming missile bogeys (air-to-air interceptor's and ground-to-air seeker's ATR processing), especially MIRVed ballistic missile warheads.

5 Appendices

5.1 VHDL code for correlation estimators

This section lists the VHDL code written to fulfill the goals of Task 1 of the contract. This listing does not include all the code written for the task, as including all the code written would have been prohibitive.

5.1.1 Polarity Coincidence Correlation Estimator:

This is the estimator requiring the least computation but offering poor accuracy. The main listing is on the next few pages. Subroutines are given at the end of this section.

```
-----------------------------------------------------------------------------
--  Implement the following function for 256 samples.          --
--                                                                                    --
--        Rij=(pi/(2*n)sum(sign(X)*sign(Y)) 1<=n<=256                   --
--                                                                                    --
--        Input is byte serial                                                        --
--                                                                                    --
-----------------------------------------------------------------------------          --

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;                                        --Mentor
Graphics Design Library

ENTITY  polcoin_s IS

        PORT    (SIGNAL xysig   :IN      SIGNED(7 DOWNTO 0);               --2s comp input,
read on the rising edge of clk.
                SIGNAL clk      :IN     STD_LOGIC;                         --clock .
        SIGNAL ce       :IN     STD_LOGIC;                             --clock enable.
        SIGNAL ares     :IN     STD_LOGIC;                             --asynchronous reset,
resets to zero.
                SIGNAL rsignal  :OUT    SIGNED(11 DOWNTO 0));                      --2s
comp output with 4 binary places, ie. pos values look like:
                                                                            --2s comp output
with 4 binary places, ie. pos values look like:
                                                                            -- [0, 2^2, 2, 1 .
2^-1, 2^-2, 2^-3, 2^-4, 2^-5, 2^-6, 2^-7, 2^-8].

ARCHITECTURE behav OF polcoin_s IS

        SIGNAL          xinput          :SIGNED(7 DOWNTO 0);
        -- Intermediate signals between the input buffer and the quadrx_p
        SIGNAL          yinput          :SIGNED(7 DOWNTO 0);
        -- component.
        SIGNAL          clk2            :STD_LOGIC;                                 --
Divide by 2 clock, transitions occur on the rising edge of clk.
        SIGNAL          clk2inv         :STD_LOGIC;                                 --
Inverse of clk2.
        SIGNAL          enclk2          :STD_LOGIC;                                 -- Clock
enable of the input buffer register that holds the y value.
        SIGNAL          enclk2inv       :STD_LOGIC;                                 -- Clock
enable of the input buffer register that holds the x value.
BEGIN
        clk2inv         <= NOT clk2;
        enclk2          <= ce AND clk2;
        enclk2inv       <= ce AND clk2inv;

clockhalf: ENTITY work.dff_pea(behav)
        PORT MAP        (q=>clk2, d=>clk2inv, clk=>clk, ares=>ares, ce=>ce);        -- Clock divider,
output is clk2 and clk2inv.
```

```
regx: ENTITY work.regs_pea(behav)                                          -- Input buffer
register that latches current value of x.
        GENERIC MAP  (width => 8)                                          -- X and y values
enter xysig, alternating in a byte serial
        PORT MAP      (q => xinput, d => xysig, clk => clk, ce => enclk2inv,-- fashion.  Regx latches
the x value on the rising edge of
                      ares => ares);                                       -- clk when ce =1
and clk2inv = 1 (clk2 =0).

regy: ENTITY work.regs_pea(behav)                                          -- Input buffer
register that latches current value of y.
        GENERIC MAP  (width => 8)                                          -- X and y values
enter xysig, alternating in a byte serial
        PORT MAP      (q => yinput, d => xysig, clk => clk, ce => enclk2,   -- fashion.  Regy latches
the y value on the rising edge o
                      ares => ares);                                       -- clk when ce =1
and clk2 = 1.

qrx: ENTITY work.polcoin_p(behav)                                          -- Quadruplex
estimator, clocked on the rising edge of clk2.
        PORT MAP      (xsignal => xinput, ysignal => yinput, clk => clk2,   -- Inputs are from the regx
and regy components.
                      ce => ce, ares => ares, rsignal=> rsignal);

END ARCHITECTURE behav;
```

```
--   Implement the following function for 256 samples.          --
--
--          Rij=(pi/(2*n)sum(sign(X)*sign(Y)) 1<=n<=256                              --
--                                                                            --
--          Input is byte parallel                                            --
--                                                                            --
-----------------------------------------------------------------------------  --

 LIBRARY ieee;
 USE ieee.std_logic_1164.ALL;

 LIBRARY arithmetic;
 USE arithmetic.std_logic_arith.ALL;                                          --Mentor
 Graphics Design Library

 ENTITY  polcoin_p IS

          PORT   (SIGNAL xsignal :IN        SIGNED(7 DOWNTO 0);               --2s comp input,
 read on the rising edge of clk.
                   SIGNAL ysignal  :IN       SIGNED(7 DOWNTO 0);              --2s comp input,
 read on the rising edge of clk.
                   SIGNAL clk      :IN     STD_LOGIC;                         --clock .
           SIGNAL ce       :IN    STD_LOGIC;                         --clock enable.
           SIGNAL ares     :IN    STD_LOGIC;                         --asynchronous reset,
 resets to zero.
                   SIGNAL rsignal  :OUT    SIGNED(11 DOWNTO 0));                      --2s
 comp output with 4 binary places, ie. pos values look like:

 2^-1, 2^-2, 2^-3, 2^-4, 2^-5, 2^-6, 2^-7, 2^-8].                             -- [0, 2^2, 2, 1 .
 END ENTITY polcoin_p;

 ARCHITECTURE behav OF polcoin_p IS

 SIGNAL          sgnx              :SIGNED(1 DOWNTO 0);
         --sign of xsignal.
 SIGNAL          sgny              :SIGNED(1 DOWNTO 0);
         --sign of ysignal.
 SIGNAL          sproduct :SIGNED(3 DOWNTO 0);                                     --
 product of signs.
 SIGNAL          sextend           :SIGNED(9 DOWNTO 0);
         --sproduct extended to a 10 bit 2s comp value.
 SIGNAL          stotal            :SIGNED(9 DOWNTO 0);
         --accumulated product of signs.
 SIGNAL          piproduct         :SIGNED(18 DOWNTO 0);
         --stotal multipled by pi.
 SIGNAL          longrsig :SIGNED(18 DOWNTO 0);                                    --
 piproduct shifted to remove non significant bits.

 BEGIN
         sgny(1)           <= ysignal(7);                                     --the MSB of sign
 is identical to the MSB of the signal.
         sgnx(1)           <= xsignal(7);
         sgny(0)           <= '0' WHEN ysignal = "00000000" ELSE '1';         --the LSB of sign
 is zero when the signal is zero.
         sgnx(0)           <= '0' WHEN xsignal = "00000000" ELSE '1';
```

```
            finput          <=ysignal WHEN crossxy='1' ELSE xsignal;
            ginput          <=xsignal WHEN crossxy='1' ELSE ysignal;

    fextend: ENTITY work.twoscomp(behav)                                --Extend the 8bit
    2s comp finput to the 16bit 2s comp fsig16bit
            GENERIC MAP  (iwidth => 8, owidth => 16)                    --and invert
    according to invertxy.
            PORT MAP        (input => finput, zero => '0', neg => invertxy,
                             output => fsig16bit);

    gextend: ENTITY work.twoscomp(behav)                                --Extend the 8bit
    2s comp ginput to the 16bit 2s comp gsig16bit
            GENERIC MAP  (iwidth => 8, owidth => 16)                    --and invert
    according to invertxy.
            PORT MAP        (input => ginput, zero => '0', neg => invertxy,
                             output => gsig16bit);

    faccum: ENTITY work.accums(behav)                                   --Accumulate the
    fsig16bit and output the 16bit 2scomp ftotal.
            GENERIC MAP  (width => 16)
            PORT MAP        (input => fsig16bit, output => ftotal, clk => clk,
                            ce =>ce, ares => ares);

    gaccum: ENTITY work.accums(behav)                                   --Accumulate the
    gsig16bit and output the 16bit 2scomp gtotal.
            GENERIC MAP  (width => 16)
            PORT MAP        (input => gsig16bit, output => gtotal, clk => clk,
                            ce =>ce, ares => ares);

            fgproduct       <= ftotal * gtotal;                     --Perform the final
    products.
            piproduct       <= fgproduct * "011001001";                     --pi*2^6 rounded
    to 6 places; pi is estimated by 11.001001 base2.
            longrsig <= piproduct / "01000000000000000000";         --Divide by 16 less
    than[((N*N*2)*(2^6)], this will leave 4
            rsignal         <= longrsig(19 DOWNTO 0);                       --places after the
    (2^0) place. Result is "[19:4].[3:0]".

    END ARCHITECTURE behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;

ENTITY  accums IS
        GENERIC
                (width:   POSITIVE :=16);
        PORT
                (SIGNAL input   :IN        SIGNED(width-1 DOWNTO 0);
                SIGNAL output  :INOUT SIGNED(width-1 DOWNTO 0);
                SIGNAL clk      :IN     STD_LOGIC;
         SIGNAL ce       :IN     STD_LOGIC;
         SIGNAL ares     :IN     STD_LOGIC);
END ENTITY accums;

ARCHITECTURE behav OF accums IS

SIGNAL          summation          :SIGNED(width-1 DOWNTO 0);

BEGIN

summation        <= output + input;

regis:    ENTITY work.regs_pea(behav)
          GENERIC MAP (width    => width)
          PORT MAP         (q =>     output, d => summation, clk => clk,
                                     ce =>ce, ares => ares);

END ARCHITECTURE behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;

ENTITY  dff_pea IS

        PORT
                (SIGNAL q       :INOUT STD_LOGIC;
                SIGNAL d        :IN     STD_LOGIC;
                SIGNAL clk      :IN     STD_LOGIC;
                SIGNAL ce       :IN     STD_LOGIC;
                SIGNAL ares     :IN     STD_LOGIC);
END ENTITY dff_pea;

ARCHITECTURE behav OF dff_pea IS

BEGIN

reg:    PROCESS (ares, clk) IS
        BEGIN
                IF ares = '1' THEN
                        q <= '0';
                ELSIF ares ='0' and (clk'event and clk ='1') THEN
                        IF ce ='1' THEN q <= d;
                        ELSE q<= q;
                        END IF;
                END IF;

        END PROCESS;

END ARCHITECTURE behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--USE ieee.std_logic_arith.ALL;
--USE ieee.std_logic_unsigned.ALL;

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;

ENTITY  regs_pea IS
        GENERIC
                (width:  POSITIVE :=16);
        PORT
                (SIGNAL q       :INOUT SIGNED(width-1 DOWNTO 0);
                SIGNAL d        :IN     SIGNED(width-1 DOWNTO 0);
                SIGNAL clk      :IN     STD_LOGIC;
                SIGNAL ce       :IN     STD_LOGIC;
                SIGNAL ares     :IN     STD_LOGIC);
END ENTITY regs_pea;

ARCHITECTURE behav OF regs_pea IS

BEGIN

reg:    PROCESS (ares, clk) IS
        BEGIN
                IF ares = '1' THEN
                        q <= sign_extend("00", width);
                ELSIF ares ='0' THEN
           IF (clk'event and clk ='1') THEN
                        IF ce ='1' THEN q <= d;
                        ELSE q<= q;
                        END IF;
           ELSE q<= q;
           END IF;
                END IF;

        END PROCESS;

END ARCHITECTURE behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;

ENTITY  twoscomp IS
        GENERIC         (iwidth:  POSITIVE :=8;
                owidth:           POSITIVE :=16);

        PORT    (SIGNAL input   :IN     SIGNED(iwidth-1 DOWNTO 0);
                 SIGNAL zero    :IN     STD_LOGIC;
                 SIGNAL neg     :IN     STD_LOGIC;
                 SIGNAL output  :OUT    SIGNED(owidth-1 DOWNTO 0));

END ENTITY twoscomp;


ARCHITECTURE behav OF twoscomp IS

BEGIN

ext:  output <=    sign_extend("00", owidth) WHEN zero='1' ELSE
                   sign_extend(-input, owidth)          WHEN zero='0' AND neg='1' ELSE
              sign_extend(input, owidth) ; -- WHEN zero='0' AND neg='0';

END ARCHITECTURE behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY correlator1 IS
     PORT (      X, Y                  : IN std_logic_vector(7 DOWNTO 0);
                                        clk,reset : IN std_logic;
                                        ROUT            : OUT std_logic_vector(31 DOWNTO 0);
                                        R                       : OUT std_logic_vector(31 DOWNTO 0);
                                        multxy          : OUT std_logic_vector(15 DOWNTO 0);
                                        Xabs,Yabs       : OUT std_logic_vector(7 DOWNTO 0);
                                        count           : OUT INTEGER;
                                        OVR                     : OUT std_logic);
END correlator1;

ARCHITECTURE structural of correlator1 IS

FUNCTION to_int(a:std_logic_vector(31 DOWNTO  0)) RETURN integer IS
                variable val: integer:= 0;
                variable b: integer:= 1;
                variable tmp : std_logic_vector(31 DOWNTO 0);
                variable neg_sign,C : std_logic;
        BEGIN
                IF (a(31) = '1') THEN
                        neg_sign := '1';
                        tmp := (NOT ( a ));
                        C:= '1';
                        FOR i IN 0 TO 31 LOOP
                                tmp(i) := tmp(i) XOR C;
                                C := tmp(i) AND C;
                        END LOOP;
                ELSE
                        neg_sign := '0';
                        tmp := a;
                END IF;

                FOR i IN 0 TO 31 LOOP
                        if (tmp(i) = '1') then
                                val := val + b;
                        end if;
                        b := b * 2;
                END LOOP;

                IF (neg_sign = '1') THEN
                        val := -val;
                END IF;

                return val;

        END to_int;


COMPONENT abs8
     PORT(  a8                          :IN  std_logic_vector(7 DOWNTO 0);
                o8                              :OUT std_logic_vector(7 DOWNTO 0));
```

```
            sproduct <= sgnx*sgny;
            sextend            <= sign_extend(sproduct, 10);

    saccum: ENTITY work.accums(behav)                                    --Accumulate the
    multiplied signed values and output the
            GENERIC MAP  (width => 10)                                   --10bit 2scomp
    stotal.
            PORT MAP        (input => sextend, output => stotal, clk => clk,
                            ce =>ce, ares => ares);

            piproduct          <= stotal * "011001001";                  --pi*2^6 rounded to 6
    places; pi is estimated by 11.001001 base2.
            longrsig <= piproduct / "010000000";                         --Divide by 256 less
    than[((N*2)*(2^6)], this will leave 8
            rsignal            <= longrsig(11 DOWNTO 0);                     --places after the
    (2^0) place. Result is "[11:8].[7:0]".

    END ARCHITECTURE behav;
```

5.1.2 Hybrid Sign Correlation Estimator

The following is the VHDL code for the hybrid sign estimator. Subroutines for this estimator are at the end of this section. The hybrid sign estimator requires more computation than the polarity coincidence but fewer than the quadruplex. The accuracy is also a compromise between the polarity coincidence and the quadruplex estimators.

```
--        Implement the following function for 256 samples.    --
--
--        R= PI*R1*R2/2                                                     --
--                                                                          --
--        R1=(1/n)*SUM(|Yn|)              1<=n<=256                         --
--                                                                          --
--        R2=(1/n)*SUM(Xn*sign(Yn))       1<=n<=256                         --
--                                                                          --
--                                                                          --
------------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;                                    --Mentor
Graphics Design Library

ENTITY  correlate2 IS

        PORT    (SIGNAL xsignal :IN      SIGNED(7 DOWNTO 0);           --2s comp input,
read on the rising edge of clk.
                 SIGNAL ysignal  :IN     SIGNED(7 DOWNTO 0);           --2s comp input,
read on the rising edge of clk.
                 SIGNAL clk     :IN    STD_LOGIC;                      --clock .
         SIGNAL ce       :IN    STD_LOGIC;                     --clock enable.
         SIGNAL ares     :IN    STD_LOGIC;                     --asynchronous reset,
resets to zero.
                 SIGNAL rsignal  :OUT    SIGNED(19 DOWNTO 0));                 --2s
comp output with 4 binary places, ie. pos values look like:

2^12, 2^11, 2^10, 2^9, 2^8, 2^7, 2^6, 2^5,                             -- [0, 2^14, 2^13,
END ENTITY correlate2;
2,1 . 2^-1, 2^-2, 2^-3, 2^-4].                                         -- 2^4, 2^3, 2^2,

ARCHITECTURE behav OF correlate2 IS

SIGNAL          ysig16bit          :SIGNED(15 DOWNTO 0);
SIGNAL          xsig16bit          :SIGNED(15 DOWNTO 0);
SIGNAL          ytotal             :SIGNED(15 DOWNTO 0);
SIGNAL          xtotal             :SIGNED(15 DOWNTO 0);
SIGNAL          xyproduct          :SIGNED(31 DOWNTO 0);
SIGNAL          piproduct          :SIGNED(40 DOWNTO 0);
SIGNAL          longrsig :SIGNED(40 DOWNTO 0);
SIGNAL          yzero              :STD_LOGIC;

BEGIN
        yzero              <= '1' WHEN ysignal="00000000" ELSE '0';        --[ysignal(7),
yzero] determine sign(Yn): [d,1]=0, [1,0]=-1, [0,0]=1.

yextend: ENTITY work.twoscomp(behav)                                   --Extend the 8bit
2s comp ysignal to the 16bit 2s comp ysig16bit.
        GENERIC MAP  (iwidth => 8, owidth => 16)
        PORT MAP         (input => ysignal, zero => '0', neg => ysignal(7),
                                        output => ysig16bit);

    xextend: ENTITY work.twoscomp(behav)                                                     --Multiply the
    8bit 2s comp xsignal by sign(Yn) and extend to the
            GENERIC MAP  (iwidth => 8, owidth => 16)                                         --16bit 2s comp
    xsig16bit.
            PORT MAP        (input => xsignal, zero => yzero, neg => ysignal(7),
                             output => xsig16bit);

    ysum: ENTITY work.accums(behav)                                                          --
    Accumulate the ysig16bit and output the 16bit 2scomp ytotal.
            GENERIC MAP  (width => 16)
            PORT MAP        (input => ysig16bit, output => ytotal, clk => clk,
                             ce =>ce, ares => ares);

    xsum: ENTITY work.accums(behav)                                                          --
    Accumulate the xsig16bit and output the 16bit 2scomp xtotal.
            GENERIC MAP  (width => 16)
            PORT MAP        (input => xsig16bit, output => xtotal, clk => clk,
                             ce =>ce, ares => ares);

            xyproduct           <= ytotal * xtotal;                                  --Perform the final
    products.
            piproduct           <= xyproduct * "011001001";                                  --pi*2^6 rounded
    to 6 places; pi is estimated by 11.001001 base2.
            longrsig <= piproduct / "010000000000000000000";                         --Divide by 16 less
    than[((N*N*2)*(2^6)], this will leave 4
            rsignal             <= longrsig(19 DOWNTO 0);                                    --places after the
    (2^0) place. Result is "[19:4].[3:0]".

    END ARCHITECTURE behav;
```

5.1.3 Quadruplex (Bit Serial) Correlation Estimator

The following is the VHDL code for the quadroplex estimator. The quadruplex is the most accurate nonlinear estimator while having less computational burden than the conventional estimator.

```
--------------------------------------------------------------------------
--   Implement the following function for 256 samples.          --
--                                                                          --
--         Rij=(pi/(2*n^2)sum(Fn)sum(Gn)     1<=n<=256                      --
--                                                                          --
--         Input is byte serial                                             --
--                                                                          --
--------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;                                  --Mentor
Graphics Design Library

ENTITY  quadrx_s IS

          PORT    (SIGNAL xysig   :IN       SIGNED(7 DOWNTO 0);               --2s comp input,
read on the rising edge of clk.
                   SIGNAL clk     :IN    STD_LOGIC;                           --clock .
          SIGNAL ce       :IN    STD_LOGIC;                               --clock enable.
          SIGNAL ares     :IN    STD_LOGIC;                               --asynchronous reset,
resets to zero.
                   SIGNAL rsignal  :OUT    SIGNED(19 DOWNTO 0));                   --2s
comp output with 4 binary places, ie. pos values look like:
                                                                      -- [0, 2^14, 2^13,
2^12, 2^11, 2^10, 2^9, 2^8, 2^7, 2^6, 2^5,
END ENTITY quadrx_s;                                                  -- 2^4, 2^3, 2^2,
2,1 . 2^-1, 2^-2, 2^-3, 2^-4].

ARCHITECTURE behav OF quadrx_s IS

          SIGNAL          xinput               :SIGNED(7 DOWNTO 0);
          -- Intermediate signals between the input buffer and the quadrx_p
          SIGNAL          yinput               :SIGNED(7 DOWNTO 0);
          -- component.
          SIGNAL          clk2                 :STD_LOGIC;                             --
Divide by 2 clock, transitions occur on the rising edge of clk.
          SIGNAL          clk2inv              :STD_LOGIC;                             --
Inverse of clk2.
          SIGNAL          enclk2               :STD_LOGIC;                             -- Clock
enable of the input buffer register that holds the y value.
          SIGNAL          enclk2inv            :STD_LOGIC;                             -- Clock
enable of the input buffer register that holds the x value.
BEGIN
          clk2inv         <= NOT clk2;
          enclk2          <= ce AND clk2;
          enclk2inv       <= ce AND clk2inv;

clockhalf: ENTITY work.dff_pea(behav)
          PORT MAP        (q=>clk2, d=>clk2inv, clk=>clk, ares=>ares, ce=>ce);        -- Clock divider,
output is clk2 and clk2inv.
```

```
regx: ENTITY work.regs_pea(behav)                                                      -- Input buffer
register that latches current value of x.
          GENERIC MAP  (width => 8)                                                    -- X and y values
enter xysig, alternating in a byte serial
          PORT MAP       (q => xinput, d => xysig, clk => clk, ce => enclk2inv,-- fashion.  Regx latches
the x value on the rising edge of
                          ares => ares);                                               -- clk when ce =1
and clk2inv = 1 (clk2 =0).

regy: ENTITY work.regs_pea(behav)                                                      -- Input buffer
register that latches current value of y.
          GENERIC MAP  (width => 8)                                                    -- X and y values
enter xysig, alternating in a byte serial
          PORT MAP       (q => yinput, d => xysig, clk => clk, ce => enclk2,     -- fashion.  Regy latches
the y value on the rising edge o
                          ares => ares);                                               -- clk when ce =1
and clk2 = 1.

qrx: ENTITY work.quadrx_p(behav)                                                       -- Quadruplex
estimator, clocked on the rising edge of clk2.
          PORT MAP       (xsignal => xinput, ysignal => yinput, clk => clk2,      -- Inputs are from the regx
and regy components.
                          ce => ce, ares => ares, rsignal=> rsignal);

END ARCHITECTURE behav;
```

```
---------------------------------------------------------------------------
-- Implement the following function for 256 samples. --
--                                                                    --
--      Rij=(pi/(2*n^2)sum(Fn)sum(Gn)    1<=n<=256                   --
--                                                                    --
--      Input is byte parallel                                        --
--                                                                    --
---------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;                              --Mentor
Graphics Design Library

ENTITY  quadrx_p IS

        PORT    (SIGNAL xsignal :IN     SIGNED(7 DOWNTO 0);          --2s comp input,
read on the rising edge of clk.
                 SIGNAL ysignal  :IN    SIGNED(7 DOWNTO 0);          --2s comp input,
read on the rising edge of clk.
                 SIGNAL clk     :IN   STD_LOGIC;                     --clock .
          SIGNAL ce      :IN    STD_LOGIC;                   --clock enable.
          SIGNAL ares    :IN    STD_LOGIC;                   --asynchronous reset,
resets to zero.
                 SIGNAL rsignal  :OUT    SIGNED(19 DOWNTO 0));                --2s
comp output with 4 binary places, ie. pos values look like:

2^12, 2^11, 2^10, 2^9, 2^8, 2^7, 2^6, 2^5,                      -- [0, 2^14, 2^13,
END ENTITY quadrx_p;
2,1 . 2^-1, 2^-2, 2^-3, 2^-4].                                  -- 2^4, 2^3, 2^2,

ARCHITECTURE behav OF quadrx_p IS


SIGNAL          finput          :SIGNED(7 DOWNTO 0);
SIGNAL          ginput          :SIGNED(7 DOWNTO 0);
SIGNAL          fsig16bit:SIGNED(15 DOWNTO 0);
SIGNAL          gsig16bit       :SIGNED(15 DOWNTO 0);
SIGNAL          ftotal          :SIGNED(15 DOWNTO 0);
SIGNAL          gtotal          :SIGNED(15 DOWNTO 0);
SIGNAL          fgproduct       :SIGNED(31 DOWNTO 0);
SIGNAL          piproduct       :SIGNED(40 DOWNTO 0);
SIGNAL          longrsig :SIGNED(40 DOWNTO 0);
SIGNAL          invertxy :STD_LOGIC;
SIGNAL          crossxy         :STD_LOGIC;

BEGIN

        crossxy         <='1' WHEN abs(xsignal) >= abs(ysignal) ELSE '0';
        invertxy <=(crossxy AND xsignal(7)) or ((NOT crossxy) AND ysignal(7));
```

```
 END COMPONENT;

 COMPONENT mult8
        PORT(  a8          :IN std_logic_vector(7 DOWNTO 0);
                           b8        :IN std_logic_vector(7 DOWNTO 0);
                           c16       :OUT std_logic_vector(15 DOWNTO 0));
 END COMPONENT;

 COMPONENT add_sub32
        PORT(  a32                    :IN std_logic_vector(31 DOWNTO 0);
                           b32                  :IN std_logic_vector(31 DOWNTO 0);
                           add_nsub     :IN  std_logic;
                           c32                  :OUT std_logic_vector(31 DOWNTO 0);
                           OVR                  :OUT std_logic);
 END COMPONENT;

 COMPONENT reg32
        PORT (  clk, reset        :IN std_logic;
                           D                    :IN std_logic_vector(31 DOWNTO 0);
                           Q                    :OUT std_logic_vector(31 DOWNTO 0));
 END COMPONENT;

 CONSTANT N                                               :INTEGER := 256;
 CONSTANT sign_ext                                   :std_logic_vector(15 DOWNTO 0):=
"0000000000000000";
 SIGNAL X_tmp,Y_tmp                                  :std_logic_vector(7 DOWNTO 0);
 SIGNAL mtmp1a                                            :std_logic_vector(15 DOWNTO 0);
 SIGNAL mtmp1b, mtmp2                                     :std_logic_vector(31 DOWNTO 0);
 SIGNAL acctmp1, accmtmp2, accmtmp3  :std_logic_vector(31 DOWNTO 0);
 SIGNAL itmp1,itmp2,itmp3                            :INTEGER;
 SIGNAL sign_out                                          :std_logic;
 SIGNAL countin, countout                            :INTEGER := 0;

 BEGIN
              sign_out <= NOT(X(7) XOR Y(7));
              abs1: abs8 PORT MAP(X, X_tmp);
              Xabs <= X_tmp;
              abs2: abs8 PORT MAP(Y, Y_tmp);
              Yabs <= Y_tmp;
              multr1: mult8 PORT MAP (X_tmp, Y_tmp, mtmp1a);
              multxy <= mtmp1a;
              mtmp1b <= sign_ext & mtmp1a;
              addr1: add_sub32 PORT MAP (accmtmp3, mtmp1b, sign_out, accmtmp2, OVR);
              reg1: reg32 PORT MAP(clk, reset, accmtmp2, accmtmp3);
              itmp1 <= to_int(accmtmp3);
              itmp2 <= itmp1 / N;
              mtmp2 <= CONV_STD_LOGIC_VECTOR(itmp2,32);
              R <= accmtmp3;
              ROUT <= mtmp2;

              cnt:PROCESS(clk,reset,countin)
                     VARIABLE countv: INTEGER;
              BEGIN
                     countv := countin;
                     if (reset = '1') then
```

```
                countv := 0;
        elsif (clk'EVENT and clk = '1') THEN
                countv := countv + 1;
        end if;
        countout <= countv;
    END PROCESS cnt;

    countin <= countout;
    count <= countout;


END structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

--hybrid sign

ENTITY correlator2 IS
     PORT (      X, Y                        : IN std_logic_vector(7 DOWNTO 0);
                          clk,reset          : IN std_logic;
                          Y_OVR, X_OVR: OUT std_logic;
                          count                    : OUT INTEGER;
                          RY,RX                    : OUT std_logic_vector(15 DOWNTO 0);
                          ROUT                     : OUT std_logic_vector(31 DOWNTO 0));
       END correlator2;

ARCHITECTURE structural of correlator2 IS

       FUNCTION to_int(a:std_logic_vector(15 DOWNTO  0)) RETURN integer IS
              variable val: integer:= 0;
              variable b: integer:= 1;
              variable tmp : std_logic_vector(15 DOWNTO 0);
              variable neg_sign,C : std_logic;
       BEGIN
              IF (a(15) = '1') THEN
                     neg_sign := '1';
                     tmp := (NOT ( a ));
                     C:= '1';
                     FOR i IN 0 TO 15 LOOP
                            tmp(i) := tmp(i) XOR C;
                            C := tmp(i) AND C;
                     END LOOP;
              ELSE
                     neg_sign := '0';
                     tmp := a;
              END IF;

              FOR i IN 0 TO 15 LOOP
                     if (tmp(i) = '1') then
                            val := val + b;
                     end if;
                     b := b * 2;
              END LOOP;

              IF (neg_sign = '1') THEN
                     val := -val;
              END IF;

              return val;

       END to_int;

COMPONENT abs8
       PORT(  a8                    :IN  std_logic_vector(7 DOWNTO 0);
                     o8                    :OUT std_logic_vector(7 DOWNTO 0));
END COMPONENT;
```

```
COMPONENT mult16
        PORT(  a16         :IN std_logic_vector(15 DOWNTO 0);
                           b16      :IN std_logic_vector(15 DOWNTO 0);
                           c32      :OUT std_logic_vector(31 DOWNTO 0));
END COMPONENT;

COMPONENT add_sub16
        PORT(  a16                       :IN std_logic_vector(15 DOWNTO 0);
                           b16                      :IN std_logic_vector(15 DOWNTO 0);
                           add_nsub           :IN  std_logic;
                           c16                      :OUT std_logic_vector(15 DOWNTO 0);
                           OVR                      :OUT std_logic);
END COMPONENT;

COMPONENT reg16
        PORT (  clk, reset          :IN std_logic;
                           D                            :IN std_logic_vector(15 DOWNTO 0);
                           Q                            :OUT std_logic_vector(15 DOWNTO 0));
END COMPONENT;

        CONSTANT N                                                        :INTEGER := 256;
        CONSTANT pi                                                       :std_logic_vector(15
DOWNTO 0):="0000000110011001";
        CONSTANT pi_shft                                          :INTEGER := 128;
        CONSTANT sign_ext                                         :std_logic_vector(7 DOWNTO
0):= "00000000";
        SIGNAL Y_tmp1, X_tmp1, X_tmp               :std_logic_vector(7 DOWNTO 0);
        SIGNAL Y_tmp2,X_tmp2,X_tmp3,Y_tmp3 :std_logic_vector(15 DOWNTO 0);
        SIGNAL Y_accum1, Y_accum2, RY_tmp   :std_logic_vector(15 DOWNTO 0);
        SIGNAL X_accum1, X_accum2, RX_tmp   :std_logic_vector(15 DOWNTO 0);
  SIGNAL Y_as                                             :std_logic;
        SIGNAL X_as                                               :std_logic := '1';
        SIGNAL itmp1,itmp2,itmp3,itmp4,itmp5,itmp6        :INTEGER;
        SIGNAL countin, countout                  :INTEGER := 0;
        SIGNAL RX_tmp_pi1                                 :std_logic_vector(31 DOWNTO 0);
        SIGNAL RX_tmp_pi2,tmp_pi                  :std_logic_vector(15 DOWNTO 0);

 BEGIN
              --Calculation of R1
              Y_as <= '1';
              abs1: abs8 PORT MAP(Y, Y_tmp1);
              Y_tmp2 <= sign_ext & Y_tmp1;
              addr1: add_sub16 PORT MAP (Y_accum2,Y_tmp2,Y_as,Y_accum1,Y_OVR);
              reg1: reg16 PORT MAP(clk, reset, Y_accum1, Y_accum2);
              itmp1 <= to_int(Y_accum2);
              itmp2 <= itmp1 / N;
              Y_tmp3 <= CONV_STD_LOGIC_VECTOR(itmp2,16);
              RY <= Y_accum2;
              RY_tmp <= Y_tmp3;

              --Calculation of R2
              abs2: abs8 PORT MAP(X, X_tmp);
        sign_y: PROCESS(Y)
        BEGIN
                if (Y = sign_ext) then
                        X_tmp1 <= sign_ext;
                else
                        X_as <= NOT(Y(7) XOR X(7));
                        X_tmp1<= X_tmp;
                end if;
        END PROCESS sign_y;

        X_tmp2 <= sign_ext & X_tmp1;
        addr2: add_sub16 PORT MAP (X_accum2,X_tmp2,X_as,X_accum1,X_OVR);
        reg2: reg16 PORT MAP(clk, reset, X_accum1, X_accum2);
        itmp3 <= to_int(X_accum2);
        itmp4<= itmp3/ N;
        X_tmp3 <= CONV_STD_LOGIC_VECTOR(itmp4,16);
        RX <= X_accum2;
        RX_tmp <= X_tmp3;

        cnt:PROCESS(clk,reset,countin)
                VARIABLE countv: INTEGER;
        BEGIN
                countv := countin;
                if (reset = '1') then
                        countv := 0;
                elsif (clk'EVENT and clk = '1') THEN
                        countv := countv + 1;
                end if;
                countout <= countv;
        END PROCESS cnt;

        countin <= countout;
        count <= countout;

        --tmp_pi <= pi;
        --multpi: mult16 PORT MAP (RX_tmp,tmp_pi,RX_tmp_pi1);
        --itmp5 <= to_int(RX_tmp_pi1(15 DOWNTO 0));
        --itmp6<= itmp5/ pi_shft;
        --RX_tmp_pi2 <= CONV_STD_LOGIC_VECTOR(itmp6,16);

        mult1: mult16 PORT MAP (RX_tmp,RY_tmp,ROUT);

END structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-polarity coin serial

ENTITY correlator3 IS
    PORT (      X, Y                    : IN std_logic_vector(7 DOWNTO 0);
                        clk,reset       : IN std_logic;
                        OVR                             : OUT std_logic;
                        count                   : OUT INTEGER;
                        ROUT                    : OUT std_logic_vector(15 DOWNTO 0));
    END correlator3;

ARCHITECTURE structural of correlator3 IS

        FUNCTION to_int(a:std_logic_vector(15 DOWNTO  0)) RETURN integer IS
                variable val: integer:= 0;
                variable b: integer:= 1;
                variable tmp : std_logic_vector(15 DOWNTO 0);
                variable neg_sign,C : std_logic;
        BEGIN
                IF (a(15) = '1') THEN
                        neg_sign := '1';
                        tmp := (NOT ( a ));
                        C:= '1';
                        FOR i IN 0 TO 15 LOOP
                                tmp(i) := tmp(i) XOR C;
                                C := tmp(i) AND C;
                        END LOOP;
                ELSE
                        neg_sign := '0';
                        tmp := a;
                END IF;

                FOR i IN 0 TO 15 LOOP
                        if (tmp(i) = '1') then
                                val := val + b;
                        end if;
                        b := b * 2;
                END LOOP;

                IF (neg_sign = '1') THEN
                        val := -val;
                END IF;

                return val;

        END to_int;

COMPONENT add_sub16
     PORT(  a16                 :IN std_logic_vector(15 DOWNTO 0);
                b16                     :IN std_logic_vector(15 DOWNTO 0);
                add_nsub        :IN  std_logic;
                c16                     :OUT std_logic_vector(15 DOWNTO 0);
```

```
                              OVR                    :OUT std_logic);
  END COMPONENT;

  COMPONENT reg16
        PORT (  clk, reset          :IN std_logic;
                              D                      :IN std_logic_vector(15 DOWNTO 0);
                              Q                      :OUT std_logic_vector(15 DOWNTO 0));
  END COMPONENT;

        CONSTANT N                                                    :INTEGER := 256;
        CONSTANT sign_ext                                        :std_logic_vector(7 DOWNTO 0)
:= "00000000";
        CONSTANT ONE                                                  :std_logic_vector(7
DOWNTO 0) := "0000000000000001";
        SIGNAL R_tmp                                              :std_logic_vector(15 DOWNTO 0);
        SIGNAL as                                                     :std_logic;
  SIGNAL countout, countin                 :INTEGER;
  SIGNAL accum1,accum2,tmp                 :std_logic_vector(15 DOWNTO 0);
        SIGNAL itmp1, itmp2                                    :INTEGER;

 BEGIN
              sign_mult: PROCESS(Y,X)
              BEGIN
                     if (Y = sign_ext) OR (X = sign_ext) then
                            R_tmp <= sign_ext & sign_ext;
                            as <= '1';
                     elsif (Y(7) = '1') AND (X(7) = '1') then
                            R_tmp <= ONE;
                            as <= '1';
                     elsif (Y(7) = '0') AND (X(7) = '0') then
                            R_tmp <= ONE;
                            as <= '1';
                     else
                            R_tmp <= ONE;
                            as <= '0';
                     end if;
              END PROCESS sign_mult;

              addr1: add_sub16 PORT MAP (accum2,R_tmp,as,accum1,OVR);
              reg1: reg16 PORT MAP(clk, reset, accum1, accum2);
              itmp1 <= to_int(accum2);
              itmp2 <= itmp1 / N;
              tmp <= CONV_STD_LOGIC_VECTOR(itmp2,16);
              ROUT <= tmp;


              cnt:PROCESS(clk,reset,countin)
                     VARIABLE countv: INTEGER;
              BEGIN
                     countv := countin;
                     if (reset = '1') then
                            countv := 0;
                     elsif (clk'EVENT and clk = '1') THEN
                            countv := countv + 1;
                     end if;
```

```
                countout <= countv;
        END PROCESS cnt;

        countin <= countout;
        count <= countout;


END structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-polarity coin parallel

ENTITY correlator3 IS
     PORT (      X1, Y1                    : IN std_logic_vector(7 DOWNTO 0);
                              X2, Y2                    : IN std_logic_vector(7 DOWNTO 0);
                              clk,reset        : IN std_logic;
                              OVR                                 : OUT std_logic;
                              count                     : OUT INTEGER;
                              ROUT                      : OUT std_logic_vector(15 DOWNTO 0));
        END correlator3;

ARCHITECTURE structural of correlator3 IS

        FUNCTION to_int(a:std_logic_vector(15 DOWNTO  0)) RETURN integer IS
                variable val: integer:= 0;
                variable b: integer:= 1;
                variable tmp : std_logic_vector(15 DOWNTO 0);
                variable neg_sign,C : std_logic;
        BEGIN
                IF (a(15) = '1') THEN
                        neg_sign := '1';
                        tmp := (NOT ( a ));
                        C:= '1';
                        FOR i IN 0 TO 15 LOOP
                                tmp(i) := tmp(i) XOR C;
                                C := tmp(i) AND C;
                        END LOOP;
                ELSE
                        neg_sign := '0';
                        tmp := a;
                END IF;

                FOR i IN 0 TO 15 LOOP
                        if (tmp(i) = '1') then
                                val := val + b;
                        end if;
                        b := b * 2;
                END LOOP;

                IF (neg_sign = '1') THEN
                        val := -val;
                END IF;

                return val;

       END to_int;

COMPONENT add_sub16
       PORT(  a16                   :IN std_logic_vector(15 DOWNTO 0);
                       b16                  :IN std_logic_vector(15 DOWNTO 0);
                       add_nsub         :IN  std_logic;
```

```
                        c16                         :OUT std_logic_vector(15 DOWNTO 0);
                        OVR                         :OUT std_logic);
  END COMPONENT;

  COMPONENT reg16
        PORT (  clk, reset          :IN std_logic;
                        D                           :IN std_logic_vector(15 DOWNTO 0);
                        Q                           :OUT std_logic_vector(15 DOWNTO 0));
  END COMPONENT;

        CONSTANT N                                                    :INTEGER := 256;
        CONSTANT sign_ext                                        :std_logic_vector(7 DOWNTO 0)
:= "00000000";
        CONSTANT sign_ext2                                       :std_logic_vector(15 DOWNTO 0)
:= "0000000000000000";
        CONSTANT ONE                                                  :std_logic_vector(15
DOWNTO 0) := "0000000000000001";
        SIGNAL R_tmp1, R_tmp2                                    :std_logic_vector(15 DOWNTO 0);
        SIGNAL as, as0, as1, as2                :std_logic;
  SIGNAL countout, countin                      :INTEGER;
  SIGNAL accum0, accum1,accum2,tmp   :std_logic_vector(15 DOWNTO 0);
        SIGNAL itmp1, itmp2                                      :INTEGER;
        SIGNAL OVR0                                                   :std_logic;

 BEGIN
                sign_mult1: PROCESS(Y1,X1)
                BEGIN
                        if (Y1 = sign_ext) OR (X1 = sign_ext) then
                                R_tmp1 <= sign_ext & sign_ext;
                                as1 <= '1';
                        elsif (Y1(7) = '1') AND (X1(7) = '1') then
                                R_tmp1 <= ONE;
                                as1 <= '1';
                        elsif (Y1(7) = '0') AND (X1(7) = '0') then
                                R_tmp1 <= ONE;
                                as1 <= '1';
                        else
                                R_tmp1 <= ONE;
                                as1 <= '0';
                        end if;
                END PROCESS sign_mult1;

        sign_mult2: PROCESS(Y2,X2)
                BEGIN
                        if (Y2 = sign_ext) OR (X2 = sign_ext) then
                                R_tmp2 <= sign_ext & sign_ext;
                                as2 <= '1';
                        elsif (Y2(7) = '1') AND (X2(7) = '1') then
                                R_tmp2 <= ONE;
                                as2 <= '1';
                        elsif (Y2(7) = '0') AND (X2(7) = '0') then
                                R_tmp2 <= ONE;
                                as2 <= '1';
                        else
                                R_tmp2 <= ONE;
                        as2 <= '0';
                end if;
        END PROCESS sign_mult2;

        as0 <= NOT( as1 XOR as2);
        as <= '1';
        addr0: add_sub16 PORT MAP (R_tmp1,R_tmp2,as0,accum0,OVR0);
        addr1: add_sub16 PORT MAP (accum2,accum0,as,accum1,OVR);
        reg1: reg16 PORT MAP(clk, reset, accum1, accum2);
        itmp1 <= to_int(accum2);
        itmp2 <= itmp1 / N;
        tmp <= CONV_STD_LOGIC_VECTOR(itmp2,16);
        ROUT <= tmp;


        cnt:PROCESS(clk,reset,countin)
                VARIABLE countv: INTEGER;
        BEGIN
                countv := countin;
                if (reset = '1') then
                        countv := 0;
                elsif (clk'EVENT and clk = '1') THEN
                        countv := countv + 1;
                end if;
                countout <= countv;
        END PROCESS cnt;

        countin <= countout;
        count <= countout;


END structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-quad serial

ENTITY correlator4 IS
    PORT (      X, Y                              : IN signed(7 DOWNTO 0);
                              clk,reset          : IN std_logic;
                              OVR                                      : OUT std_logic;
                              --count                      : OUT INTEGER;
                              f_tmp1,g_tmp1   : OUT signed(7 DOWNTO 0);
                              f_out,g_out                  : OUT INTEGER;
                              ROUT                         : OUT std_logic_vector(31 DOWNTO 0));
        END correlator4;

ARCHITECTURE structural of correlator4 IS


FUNCTION to_int16(a:std_logic_vector(15 DOWNTO  0)) RETURN integer IS
                variable val: integer:= 0;
                variable b: integer:= 1;
                variable tmp : std_logic_vector(15 DOWNTO 0);
                variable neg_sign,C : std_logic;
        BEGIN
                IF (a(15) = '1') THEN
                        neg_sign := '1';
                        tmp := (NOT ( a ));
                        C:= '1';
                        FOR i IN 0 TO 15 LOOP
                                tmp(i) := tmp(i) XOR C;
                                C := tmp(i) AND C;
                        END LOOP;
                ELSE
                        neg_sign := '0';
                        tmp := a;
                END IF;

                FOR i IN 0 TO 15 LOOP
                        if (tmp(i) = '1') then
                                val := val + b;
                        end if;
                        b := b * 2;
                END LOOP;

                IF (neg_sign = '1') THEN
                        val := -val;
                END IF;

                return val;

        END to_int16;

COMPONENT mult16
        PORT(  a16       :IN std_logic_vector(15 DOWNTO 0);
                b16       :IN std_logic_vector(15 DOWNTO 0);
                c32       :OUT std_logic_vector(31 DOWNTO 0));
 END COMPONENT;

 COMPONENT add_sub16
        PORT(  a16, b16:IN  std_logic_vector(15 DOWNTO 0);
                add_nsub          :IN  std_logic;
                c16                         :OUT std_logic_vector(15 DOWNTO 0);
                OVR                         :OUT std_logic);
 END COMPONENT;

 COMPONENT reg16
        PORT (  clk, reset         :IN std_logic;
                D                           :IN std_logic_vector(15 DOWNTO 0);
                Q                           :OUT std_logic_vector(15 DOWNTO 0));
 END COMPONENT;

 COMPONENT abs16
        PORT(  a16                     :IN  std_logic_vector(15 DOWNTO 0);
                o16                         :OUT std_logic_vector(15 DOWNTO 0));
 END COMPONENT;

        CONSTANT N                                                  :INTEGER := 256;
        SIGNAL countin, countout                       :INTEGER := 0;
        SIGNAL f_accum1,f_accum2, f_accum3, f_tmp            :std_logic_vector(15 DOWNTO 0);
        SIGNAL g_accum1,g_accum2, g_accum3, g_tmp            :std_logic_vector(15 DOWNTO 0);
        SIGNAL f_accum, g_accum                                   :std_logic_vector(15 DOWNTO 0);
        SIGNAL X_int, Y_int, R_int                     :INTEGER;
 SIGNAL f_int, f_int1, f_int2, f_int3       :INTEGER;
        SIGNAL g_int, g_int1, g_int2, g_int3       :INTEGER;
        SIGNAL asf, asg,  OVRF, OVRG              :std_logic;


BEGIN
            quad_func: PROCESS(X,Y)
                  VARIABLE tmpf, tmpg : signed(7 DOWNTO 0);
                  VARIABLE xin, xin_n, yin, yin_n : signed(7 DOWNTO 0);
                  VARIABLE xin_t, yin_t : integer;
            BEGIN
                  xin        := X;
                  xin_t := CONV_INTEGER(X);
                  xin_n := CONV_SIGNED(-xin_t,8);
                  yin        := Y;
                  yin_t  := CONV_INTEGER(Y);
                  yin_n := CONV_SIGNED(-yin_t,8);
                  if ( xin_n <= yin ) AND ( yin <= xin) then
                        tmpf := yin;
                        tmpg := xin;
                  elsif  ( yin_n <= xin ) AND ( xin <= yin) then
                        tmpf := xin;
                        tmpg := yin;
                  elsif  ( xin_n >= yin ) AND ( yin >= xin) then
                        tmpf := yin_n;
                        tmpg := xin_n;
                  elsif  ( yin_n >= xin ) AND ( xin >= yin) then
                tmpf := xin_n;
                tmpg := yin_n;
        end if;

        f_tmp1 <= tmpf;
        g_tmp1 <= tmpg;

        f_accum <= CONV_STD_LOGIC_VECTOR(tmpf,16);
        g_accum <= CONV_STD_LOGIC_VECTOR(tmpg,16);
END PROCESS quad_func;

asf <= NOT(f_accum(15));
asg <= NOT(g_accum(15));

abs1: abs16 PORT MAP (f_accum, f_accum3);
abs2: abs16 PORT MAP (g_accum, g_accum3);

addf: add_sub16 PORT MAP(f_accum2, f_accum3, asf, f_accum1, OVRF);
addg: add_sub16 PORT MAP(g_accum2, g_accum3, asg, g_accum1, OVRG);

reg1: reg16 PORT MAP(clk, reset, f_accum1, f_accum2);
reg2: reg16 PORT MAP(clk, reset, g_accum1, g_accum2);

f_int2 <= to_int16(f_accum2);
g_int2 <= to_int16(g_accum2);

f_int3 <= f_int2 / N;
g_int3 <= g_int2 / N;

f_out <= f_int3;
g_out <= g_int3;

f_tmp <= CONV_STD_LOGIC_VECTOR(f_int3,16);
g_tmp <= CONV_STD_LOGIC_VECTOR(g_int3,16);

OVR <= OVRF OR OVRG;
mult1: mult16 PORT MAP (f_tmp,g_tmp,ROUT);


--cnt:PROCESS(clk,reset,countin)
--        VARIABLE countv: INTEGER;
--BEGIN
--        countv := countin;
          --if (reset = '1') then
          --        countv := 0;
          --elsif (clk'EVENT and clk = '1') THEN
          --        countv := countv + 1;
--end if;
--        countout <= countv;
--END PROCESS cnt;

--countin <= countout;
--count <= countout;
```

```
END structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-quad parallel

ENTITY correlator4 IS
    PORT (      X1, Y1                      : IN signed(7 DOWNTO 0);
                            X2, Y2                      : IN signed(7 DOWNTO 0);
                            clk,reset           : IN std_logic;
                            OVR                                 : OUT std_logic;
                            f_tmp1,g_tmp1   : OUT std_logic_vector( 15 DOWNTO 0);
                            f_out,g_out                 : OUT INTEGER;
                            ROUT                        : OUT std_logic_vector(31 DOWNTO 0));
        END correlator4;

ARCHITECTURE structural of correlator4 IS

FUNCTION sign_ext(a:signed(7 DOWNTO 0)) RETURN signed IS
                VARIABLE tmp: signed(15 DOWNTO 0);
        BEGIN
                if (a(7) = '1') then
                        tmp := "11111111" & a;
                else tmp := "00000000" & a;
                end if;
                return tmp;
        END sign_ext;

FUNCTION to_int16(a:std_logic_vector(15 DOWNTO  0)) RETURN integer IS
                variable val: integer:= 0;
                variable b: integer:= 1;
                variable tmp : std_logic_vector(15 DOWNTO 0);
                variable neg_sign,C : std_logic;
        BEGIN
                IF (a(15) = '1') THEN
                        neg_sign := '1';
                        tmp := (NOT ( a ));
                        C:= '1';
                        FOR i IN 0 TO 15 LOOP
                                tmp(i) := tmp(i) XOR C;
                                C := tmp(i) AND C;
                        END LOOP;
                ELSE
                        neg_sign := '0';
                        tmp := a;
                END IF;

                FOR i IN 0 TO 15 LOOP
                        if (tmp(i) = '1') then
                                val := val + b;
                        end if;
                        b := b * 2;
                END LOOP;

                IF (neg_sign = '1') THEN
```

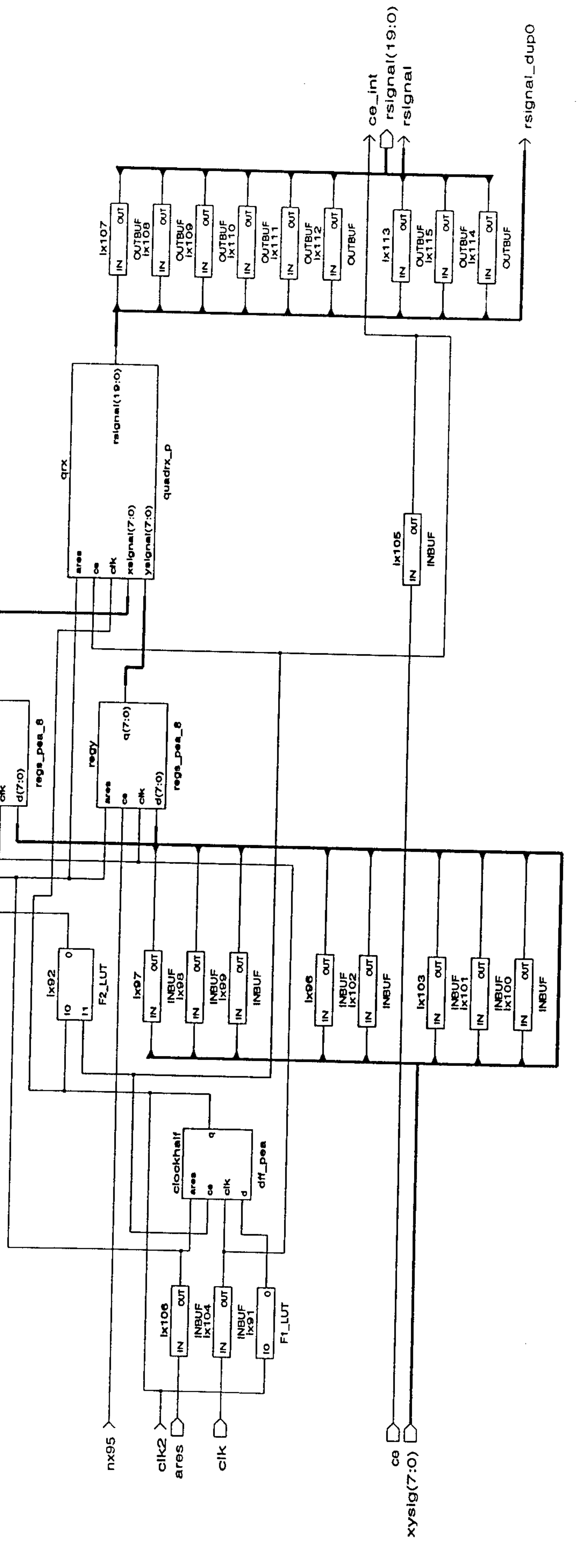
```
                  val := -val;
              END IF;

              return val;

          END to_int16;

      COMPONENT mult16
          PORT(  a16       :IN std_logic_vector(15 DOWNTO 0);
                           b16       :IN std_logic_vector(15 DOWNTO 0);
                           c32       :OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      COMPONENT add_sub16
          PORT(  a16, b16:IN  std_logic_vector(15 DOWNTO 0);
                           add_nsub          :IN  std_logic;
                           c16                         :OUT std_logic_vector(15 DOWNTO 0);
                           OVR                         :OUT std_logic);
      END COMPONENT;

      COMPONENT reg16
          PORT (  clk, reset          :IN std_logic;
                           D                           :IN std_logic_vector(15 DOWNTO 0);
                           Q                           :OUT std_logic_vector(15 DOWNTO 0));
      END COMPONENT;

      COMPONENT abs16
          PORT(  a16                      :IN  std_logic_vector(15 DOWNTO 0);
                           o16                         :OUT std_logic_vector(15 DOWNTO 0));
      END COMPONENT;

          CONSTANT N                                                              :INTEGER := 256;
          SIGNAL countin, countout                                  :INTEGER := 0;
          SIGNAL f_accum1,f_accum2, f_accum3, f_tmp                  :std_logic_vector(15 DOWNTO 0);
          SIGNAL f_accumt2, g_accumt2, g_accumh2, f_accumh2                   :std_logic_vector(15
DOWNTO 0);
          SIGNAL g_accum1,g_accum2, g_accum3, g_tmp, g_accumh, g_accuml
          :std_logic_vector(15 DOWNTO 0);
          SIGNAL f_accum, g_accum, f_accumt, g_accumt, f_accumh, f_accuml :std_logic_vector(15
DOWNTO 0);
          SIGNAL X_int, Y_int, R_int                               :INTEGER;
   SIGNAL f_int, f_int1, f_int2, f_int3         :INTEGER;
          SIGNAL g_int, g_int1, g_int2, g_int3          :INTEGER;
          SIGNAL asf1, asg1, asf2, asg2, OVRF, OVRG, OVRFt, OVRGt                    :std_logic;


 BEGIN
              quad_func1: PROCESS(X1,Y1)
                  VARIABLE tmpf, tmpg : signed(7 DOWNTO 0);
                  VARIABLE f_accumlt, g_accumlt : signed(15 DOWNTO 0);
                  VARIABLE xin, xin_n, yin, yin_n : signed(7 DOWNTO 0);
                  VARIABLE xin_t, yin_t : integer;
              BEGIN
                  xin          := X1;
                  xin_t := CONV_INTEGER(X1);
```

```
                        xin_n := CONV_SIGNED(-xin_t,8);
                        yin        := Y1;
                        yin_t  := CONV_INTEGER(Y1);
                        yin_n := CONV_SIGNED(-yin_t,8);
                        if ( xin_n <= yin ) AND ( yin <= xin) then
                                tmpf := yin;
                                tmpg := xin;
                        elsif  ( yin_n <= xin ) AND ( xin <= yin) then
                                tmpf := xin;
                                tmpg := yin;
                        elsif  ( xin_n >= yin ) AND ( yin >= xin) then
                                tmpf := yin_n;
                                tmpg := xin_n;
                        elsif  ( yin_n >= xin ) AND ( xin >= yin) then
                                tmpf := xin_n;
                                tmpg := yin_n;
                        end if;

                        f_accumlt := sign_ext(tmpf);
                        f_accuml <= CONV_STD_LOGIC_VECTOR(f_accumlt,16);
                        g_accumlt := sign_ext(tmpg);
                        g_accuml <= CONV_STD_LOGIC_VECTOR(g_accumlt,16);
                END PROCESS quad_func1;

        quad_func2: PROCESS(X2,Y2)
                        VARIABLE tmpf, tmpg : signed(7 DOWNTO 0);
                        VARIABLE f_accumht, g_accumht : signed(15 DOWNTO 0);
                        VARIABLE xin, xin_n, yin, yin_n : signed(7 DOWNTO 0);
                        VARIABLE xin_t, yin_t : integer;
                BEGIN
                        xin        := X2;
                        xin_t := CONV_INTEGER(X2);
                        xin_n := CONV_SIGNED(-xin_t,8);
                        yin        := Y2;
                        yin_t  := CONV_INTEGER(Y2);
                        yin_n := CONV_SIGNED(-yin_t,8);
                        if ( xin_n <= yin ) AND ( yin <= xin) then
                                tmpf := yin;
                                tmpg := xin;
                        elsif  ( yin_n <= xin ) AND ( xin <= yin) then
                                tmpf := xin;
                                tmpg := yin;
                        elsif  ( xin_n >= yin ) AND ( yin >= xin) then
                                tmpf := yin_n;
                                tmpg := xin_n;
                        elsif  ( yin_n >= xin ) AND ( xin >= yin) then
                                tmpf := xin_n;
                                tmpg := yin_n;
                        end if;

                        f_accumht := sign_ext(tmpf);
                        f_accumh <= CONV_STD_LOGIC_VECTOR(f_accumht,16);
                        g_accumht := sign_ext(tmpg);
                        g_accumh <= CONV_STD_LOGIC_VECTOR(g_accumht,16);
                END PROCESS quad_func2;
```

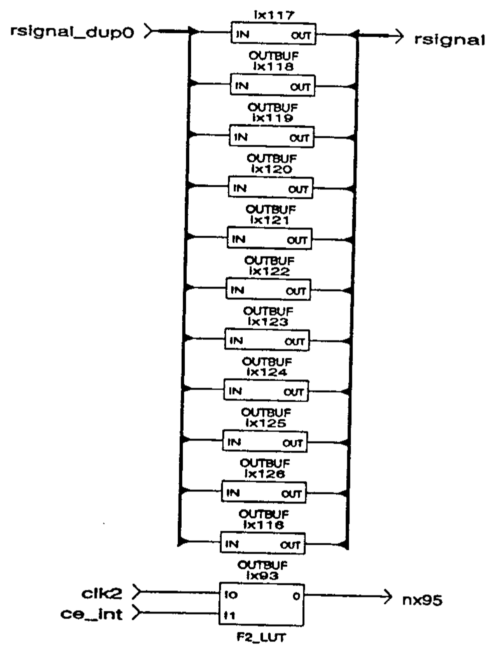
```
                    asf1 <= NOT(f_accumh(15));
                    asg1 <= NOT(g_accumh(15));

                    f_tmp1 <= f_accumh;
                    g_tmp1 <= g_accumh;

                    abs1 : abs16 PORT MAP(f_accumh, f_accumh2);
                    abs2 : abs16 PORT MAP(g_accumh, g_accumh2);
                    addft: add_sub16 PORT MAP(f_accuml, f_accumh2, asf1, f_accumt, OVRFt);
                    addgt: add_sub16 PORT MAP(g_accuml, g_accumh2, asg1, g_accumt, OVRGt);

                    --f_tmp1 <= f_accumt;
                    --g_tmp1 <= g_accumt;

                    asf2 <= NOT(f_accumt(15));
                    asg2 <= NOT(g_accumt(15));
                    abs3 : abs16 PORT MAP (f_accumt, f_accumt2);
                    abs4 : abs16 PORT MAP (g_accumt, g_accumt2);

                    addf: add_sub16 PORT MAP(f_accum2, f_accumt2, asf2, f_accum1, OVRF);
                    addg: add_sub16 PORT MAP(g_accum2, g_accumt2, asg2, g_accum1, OVRG);

                    reg1: reg16 PORT MAP(clk, reset, f_accum1, f_accum2);
                    reg2: reg16 PORT MAP(clk, reset, g_accum1, g_accum2);

                    f_int2 <= to_int16(f_accum2);
                    g_int2 <= to_int16(g_accum2);

                    f_int3 <= f_int2 / N;
                    g_int3 <= g_int2 / N;

                    f_out <= f_int3;
                    g_out <= g_int3;

                    f_tmp <= CONV_STD_LOGIC_VECTOR(f_int3,16);
                    g_tmp <= CONV_STD_LOGIC_VECTOR(g_int3,16);

                    OVR <= OVRF OR OVRG;
                    mult1: mult16 PORT MAP (f_tmp,g_tmp,ROUT);

--
--
--
--
  END structural;
```

5.2 Schematics for correlation estimators

5.2.1 Quadruplex (Bit Serial) Correlation Estimator



nx95
clk2
ares
clk
ce
xysig(7:0)
lx106
INBUF
lx104
INBUF
lx91
F1_LUT
clockhalf
dff_pea
lx92
F2_LUT
lx97
INBUF
lx98
INBUF
lx99
INBUF
lx96
INBUF
lx102
INBUF
lx103
INBUF
lx101
INBUF
lx100
INBUF
regx
regs_pea_8
regy
regs_pea_8
qrx
quadrx_p
lx105
INBUF
lx107
OUTBUF
lx108
OUTBUF
lx109
OUTBUF
lx110
OUTBUF
lx111
OUTBUF
lx112
OUTBUF
lx113
OUTBUF
lx115
OUTBUF
lx114
OUTBUF
ce_int
rsignal(19:0)
rsignal
rsignal_dup0


rsignal_dup0
rsignal
lx117
OUTBUF lx118
OUTBUF lx119
OUTBUF lx120
OUTBUF lx121
OUTBUF lx122
OUTBUF lx123
OUTBUF lx124
OUTBUF lx125
OUTBUF lx126
OUTBUF lx116
OUTBUF lx93
IN
OUT
clk2
ce_int
I0
I1
O
nx95
F2_LUT

6 References

[1] M. Sullivan and E. Wegman, "Estimating spectral correlations with simple nonlinear transformations," *IEEE Transactions on Signal Processing*, pp. 1525-1526, June 1995.

[2] M. Sullivan. "Multiplier shortcuts and FPGA-based DSP," *IEEE Signal Processing Magazine*, in review.

[3] M. Sullivan and E. Wegman, "A normalized correlation estimator for complex data based on a quadruplex transformation," to appear in *IEEE Signal Processing Letters*, January 1997.

[4] M. Sullivan and E. Wegman, "Correlation estimators based on simple nonlinear transformations," *IEEE Transactions on Signal Processing*, pp. 1438-1444, June 1995.

[5] M. Sullivan, "A signed maximum correlation multiplier for LMS filter adaptation," *IEEE Transactions on Signal Processing*, pp. 147-151, January 1993.

[6] M. Sullivan, "Efficient autocorrelation estimation using relative magnitudes," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, pp. 445-447, March 1989.

[7] K.J. Page, J. Arrigo, P. M. Chau, "ReConfigurable Hardware based Digital Signal Processing for Wireless Communications", *Proceedings of the SPIE: Advanced Signal Processing Algorithms, Architectures and Implementations VIII*, July 1997.

[8] K.J. Page and P.M. Chau, "Folding Large Regular Computational Graphs onto Smaller Processor Arrays," *Proceedings of the SPIE: Advanced Signal Processing Algorithms, Architectures and Implementations VII*, Vol. 2846, pp. 383-394, Aug. 1996.

[9] K.J. Page and P.M. Chau, "Index Mapping for Reconfigurable Communications Architectures," *4th Reconfigurable Architectures Workshop (RAW-97) is part of the 11th International Parallel Processing Symposium (IPPS-97)*, held April 1 - 5, 1997 at University of Geneva, Switzerland.

[10] Geisel, William, "Tutorial on Reed-Solomon Error Correction Coding", *NASA Technical Memorandum 10262*, August 1990.

[11] Heather Bowers, Hui Zhang, "Comparison of Reed Solomon Codec Implementations", *University of California at Berkeley CS252 Project Report*, December 1996.

[12] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error Correcting Coding: Turbo Codes." Proc. 1993 IEEE International Conference on Communications, pp. 1064-1070.

[13] http://www331.jpl.nasa.gov/public/TurboForce.GIF.

[14] D. Divsalar, and F. Pollara, "Turbo Codes for PCS Applications", 1995 *Proceedings IEEE International Conference on Communications*, pp. 54-9.

[15] A. Sarajedini and P.M. Chau, "Fast blind signal separation for multi-user interference rejection in smart antenna arrays," *Proceedings of the SPIE: Advanced Signal Processing Algorithms, Architectures and Implementations VIII*, (to be published) July 1997.

[16] R. Muira, T. Tanaka, I. Chiba, A. Horie, and Y. Karasawa, "Beamforming experiment with a DBF multibeam antenna in a mobile satellite environment," *IEEE Trans. on Antennas and Propagation*, April 1997, vol.45 (no.4):707-713.

[17] A.J. Bell and T.J. Sejnowski, "An information-maximum approach to blind seperation and blind deconvolution," *Neural Computation*, 1995, vol.7:1129-1159.

[18] Y.J. Wu and P.M. Chau, "Artificial intelligent adaptive control for DS-CDMA with open-loop power control operating over a low Earth orbiting satellite link," *Proceedings of MILCOM 1996,* vol. 1, pp. 1-5, held Oct. 22 - 24, 1996 at McLean, VA.

[19] M.J. Wirthlin, B.L. Hutchings, "Sequencing run-time reconfigured hardware with software," *ACM/SIGDA Intl. Symp. on FPGAs*, pp. 122-128, 1996.

[20] S. Dutt and F. Hanchek, "REMOD: A new methodology for designing fault-tolerant arithmetic circuits," *IEEE Trans. on VLSI Systems*, March 1996, vol.4(no.1):56-69.

[21] S. Hauck, "The roles of FPGAs in reprogrammable systems," *submitted to Proceedings of the IEEE*, 1997.

[22]Jacovitti, G., Neri, A. and R. Cusani, "Methods for estimating the autocorrelation function of complex stationary Gaussian processes." IEEE Trans. Acoustic, Speech, Signal Processing, pp. 1126-1138. Aug. 1987.

[23] Johnson, D.H. and D. Dudgeon, 'Array Signal Processing: Concepts and Techniques', Englewood Cliffs, NJ: Prentice Hall, 1993.